Zeebo Developer Guide

Gaming for the next billion

An in-depth guide for the Zeebo wireless gaming platform

Zeebo Inc. Confidential and Proprietary

Restricted Distribution:

Not to be used, copied, reproduced in whole or in part, nor its contents revealed in any manner to others without the express written permission of Zeebo Inc.

Zeebo Inc reserves the right to make changes to the product(s) or information contained herein without notice. No liability is assumed for any damages arising directly or indirectly by their use or application. The information provided in this document is provided on an “as is” basis.

This document contains Zeebo Inc confidential and proprietary information and must be shredded when discarded.

Zeebo is a registered trademark and registered service mark of Zeebo Inc. Other product and brand names may be trademarks or registered trademarks of their respective owners. CDMA2000 and GSM are registered certification mark of the Telecommunications Industry Association, used under license. ARM is a registered trademark of ARM Limited. Adreno, QXEngine, QXProfiler, BREW are registered trademarks of QUALCOMM Incorporated in the United States and other countries.

Contents

1 Introduction **6**
1.1 Purpose **6**
1.2 Scope **6**
1.3 Conventions **6**
1.4 Revision history **6**
1.5 References **7**
1.6 Technical assistance **7**
1.7 Acronyms **7**

2 Zeebo Overview **8**
2.1 Zeebo Technical Specifications **8**
2.2 User Interface **10**
2.2.1 User Interface Navigation 10
2.3 Game File Size **12**
2.3.1 Low Price Category 12
2.3.2 Standard Price Category 12
2.3.3 Premium Pricing Category 13
2.4 Tools **13**
2.4.1 QXEngine 13
2.4.2 Adreno Profiler 13

3 Zeebo System Architecture **15**
3.1 Hardware Architecture **15**
3.1.1 Memory Interface 16
3.1.2 Rendering Flow 17
3.1.3 Binning 18
3.1.4 Ring Buffer 19
3.1.5 Power Management 19
3.1.6 Display Support 20
3.2 Software Architecture **20**
3.2.1 3D Graphics Architecture 20
3.2.2 Fixed-point Math Support 21

4 Operating Systems Overview **24**
4.1 Creating and releasing BREW object instances **24**
4.2 Event Handling **24**
4.3 Cross Platform Programming **25**
4.4 Timers **26**
4.5 Multitasking **27**
4.6 Debugging **27**

5 Basic Memory Management **29**
5.1 File System, Heap and Stack Sizes **29**
5.2 Memory Alignment Issues on ARM Processors **29**
5.2.1 Symptoms 30
5.2.2 Common Causes 30

5.2.3 Recommended Solutions 31
5.2.4 Testing With the BREW Simulator 33

6 Input/Output 34
6.1 Understanding the Zeebo Gamepads 34
6.2 IHID Overview 34
6.2.1 Using ISignal 34
6.2.2 Using IHID to determine which devices are attached 35
6.3 Understanding System I/O 37
6.3.1 Creating a IHIDDevice reference 37
6.3.2 Button Events 37
6.3.3 Axis Events 37
6.3.4 Device Events 38
6.3.5 Gamepad Rumble 38
6.3.6 Exclusive Access 38
6.3.7 Default Event Handling 39
6.4 Zeebo Gamepad Remapping 39

7 3D Graphics 41
7.1 OpenGL ES Overview 41
7.1.1 Frame Buffer (Color) 41
7.1.2 Color Buffer Extension 41
7.1.3 Textures 41
7.1.4 Back-face Culling 42
7.1.5 Fog 42
7.1.6 Clearing the Color Buffer 42
7.1.7 Batching 42
7.1.8 Stencil 43
7.1.9 Blending 43
7.1.10 Lightning 43
7.1.11 Data Types and Precision 43
7.1.12 Extended Data Types 44
7.2 Supported OpenGL ES extensions 44
7.3 Using OpenGL ES and EGL APIs in BREW 45
7.3.1 Steps for using standard OpenGL ES API 45
7.4 Supported 3D Graphics API 46
7.4.1 Summary of 3D Accelerated Rendering Support 46
7.4.2 3D Graphics Limitations 47

8 3D Graphics Optimization and Tuning 48
8.1 Working with Vertex Buffer Objects (VBO's) 48
8.1.1 Preparing the VBO function extensions 48
8.1.2 Initializing a VBO 49
8.1.3 Drawing with a VBO 49
8.2 Texture Compression 50
8.2.1 QXTextureConverter 50
8.2.2 The Compressonator 51
8.2.3 ATI Compress 51
8.3 Rescaling OpenGL ES Rendering Surfaces 51

9 Zeebo Audio 54
9.1 Playback of MIDI and encoded audio objects 54

9.1.1 Supported encoded audio formats ... 54
9.1.2 Supported MIDI file formats ... 56
9.2 Multisequencing – Simultaneous playback of audio objects ... 58
9.2.1 Restrictions with MP3 and simultaneous audio ... 58
9.2.2 Exceeding number of resources with simultaneous audio ... 59
9.2.3 Decoder allocation algorithm with simultaneous audio ... 59
9.3 QAudioFX – 3D audio ... 63
9.3.1 Audio objects and environments ... 63
9.3.2 Positional audio and rolloff effects ... 64
9.3.3 Reverberation effects ... 64
9.3.4 Doppler effects ... 65
9.4 Summary of audio support on Zeebo ... 65
9.5 BREW APIs for Audio ... 66
9.5.1 Playing multiple audio objects simultaneously ... 66
9.5.2 Global Loading and Unloading of DLS ... 70
9.5.3 Sending MIDI Messages ... 72
9.5.4 QAudioFX – 3D Audio ... 75

10 Zeebo.lib System Library ... 88
10.1 Suspend and Resume ... 88
10.2 Virtual Keyboard ... 88
10.3 Controller Discovery and Removal ... 89

11 Additional Information and Requirements ... 90
11.1 Compiling Zeebo Games ... 90
11.2 Tuning for TV-Out ... 90
11.3 Saving data to console ... 90
11.4 Using BREW AppLoader ... 91
11.5 Using BREW Logger ... 95
11.6 Understanding USB Download and Trace mode ... 97
11.7 Uploading games using SD card ... 99
11.8 Power Button Behavior ... 100
11.9 Known Issues ... 100

12 Submission process ... 102

Appendix A- Supported BREW API List ... 103

Appendix B – Supported OpenGL ES API List ... 107

Appendix C – List of Acronyms ... 119

1 Introduction

1.1 Purpose

This developer guide is intended to provide information for developing and optimizing game content for the Zeebo wireless gaming platform.

1.2 Scope

This guide only covers software version 1.0 of the Zeebo wireless gaming platform, which is based off of Qualcomm's BREW 4.0.2 SDK. It also covers the usage of OpenGL ES and IHID extensions. Qualcomm's BREW 4.0.3 SDK, a new release version of BREW SDK, can also be used as a base development kit.

1.3 Conventions

Function declarations, function names, type declarations, and code samples appear in a different font, e.g., `#include`.

1.4 Revision history

The revision history for this document is shown in Table 1-1.

Table 1-1 Revision history

Version	Date	Description
Draft	April 2008	Initial release
0.7	May 2008	Updated draft
0.8	June 2008	Updated audio and BREW API sections
0.9	August 2008	Updated Zeebo.lib and 3D rendering rescaling
0.92	October 2008	Updated Zpad Remapping and multiple sounds
0.95	January 2009	Updated Known Issues and limitations
0.97	May 2009	Sections review

Zeebo Confidential and Proprietary

1.5 References

Reference documents are listed in Table 1-2. Reference documents that are no longer applicable are deleted from this table. Therefore, reference numbers may not be sequential.

Table 1-2 Reference documents

Reference	Document	
R1	OpenGL ES 1.0 Specification	http://www.khronos.org/opengles/spec
R2	OpenGL ES Game Development (book)	ISBN-10: 1592003702

1.6 Technical assistance

For assistance or clarification on information in this guide, submit questions to Zeebo developer support at devsupport@zeeboinc.com.

1.7 Acronyms

For definitions of terms and abbreviations, refer to Appendix C.

Zeebo Confidential and Proprietary

2 Zeebo Overview

2.1 Zeebo Technical Specifications

- Form-factor: Upright Standing Compact Video Game Console;
- Dimensions: 157 x 215.4 x 44 mm (L x W x D);
- Weight: 3 pounds or 1.3KG;
- Chipset:
    - MSM7201A;
    - RTR6280;
    - PM7540;
- Applications Processing (Audio/Graphics):
    - ARM 11 / QDSP-5 running at 528Mhz;
    - Audio/Graphics Processing: ARM 11 / QDSP-5 running at 528Mhz;
    - 3D Graphics Processing:
        - Qualcomm Adreno 130 Graphics Core;
        - Mobile Display Processor (MDP) (Concurrent 3D Rendering & Screen Operations);
- Internal Memory:
    - ROM: 1 GByte NAND Flash;
    - RAM: 128 MBytes DDR SDRAM + 32Mbyte stacked DDR SDRAM in MSM7201A;
- Polygon performance: 1.6M triangles per second;
- 3D pixel Fill Rate (textured): 63M polygons (2 textures);
- Output Color System: PAL-M and NTSC;
- Video-Out resolution: VGA (640X480) - 4:3 aspect ratio;
- Banding:
    - Quad Band GSM/GPRS/EDGE (850/900/1800/1900)MHz;
    - Tri Band UMTS/HSDPA/HSUPA (850/1900/2100)MHz;
- Antenna: Internal;

- Modem Processing (Integrated in MSM7201A): ARM 9 CPU / QDSP-4;
- Power: Internal AC Power Adapter with External Wall Connected Power Cord (Battery Not Required);
- Power supply: 100 – 240 V (50-60Hz) (universal);
- I/O Connectors:
    - USB 2.0 Standard A (Accessories);
    - USB 2.0 Standard A (Accessories);
    - USB 2.0 Standard A (Accessories);
    - USB 2.0 OTG – Mini B (Accessible to Service Centers and Developers only);
    - RCA Connector (Video Composite TV Signal, x2 Stereo Audio);
    - SD Card Slot/Interface;
- Power Key Functions: Power On / Power Down / Power Off;
- External Light Bar (Blue LED) Functions:
    - Console on "power off" mode: LED is turned off (button press >5seconds);
    - Console on "power down" mode: LED is presented with half brightness;
    - Console on "Power On" mode: LED is on;
    - Console on "Sleep Mode" (Screen Saver): LED is slow pulsing (increasing the brightness to 100% and decreasing to 0% in about 3 seconds);
    - Console receiving OTA updates / Receiving Data / Download: LED is fast pulsing (increasing the brightness to 100% and decreasing to 0% in about 1 seconds);
- Operating System: BREW 4.0.2;
- Pre-Loaded content: 4 BREW games + 1 Free OTA Game;
- 3D Rendering API: OpenGL ES 1.0+ Common Profile;
- 2D Rendering API (BREW 2D API);
- Audio Formats:
    - MIDI (72 voice polyphony 512 kB wavetable, 44kHz sampling rate);
    - MP3;
    - PCM;
    - ADPCM;
    - CMX;
    - QCELP;
- Messaging: SMS is server initiated only;

- UI: Proprietary Zeebo User Interface (Updatable OTA);
- Console ID: International Mobile Equipment Identity – IMEI;

2.2 User Interface

The user interface design seen in Zeebo is an interpretation of a real world retail experience for games. We have taken familiar nuances found in retail and have extended those themes into the structure, look, and logic of the interface.

The console is designed to be connected via standard RCA cables to a non-high definition color television, which is commonly found in developing countries of the world.

Updates will be made available only on certain screens and only certain elements within those screens.

Each time the console is turned on there will be a sound effect and the Zeebo image will flash and be illuminated. The interface may include other sound effects as well.

2.2.1 User Interface Navigation

During the UI navigation, the user will navigate using the buttons A, B1, C (1-2-3-4) and E (Home). Buttons B2, D1 and D2 will be enabled only for game play, should the game require them. Figure 2-1 illustrates the Zeebo gamepad.

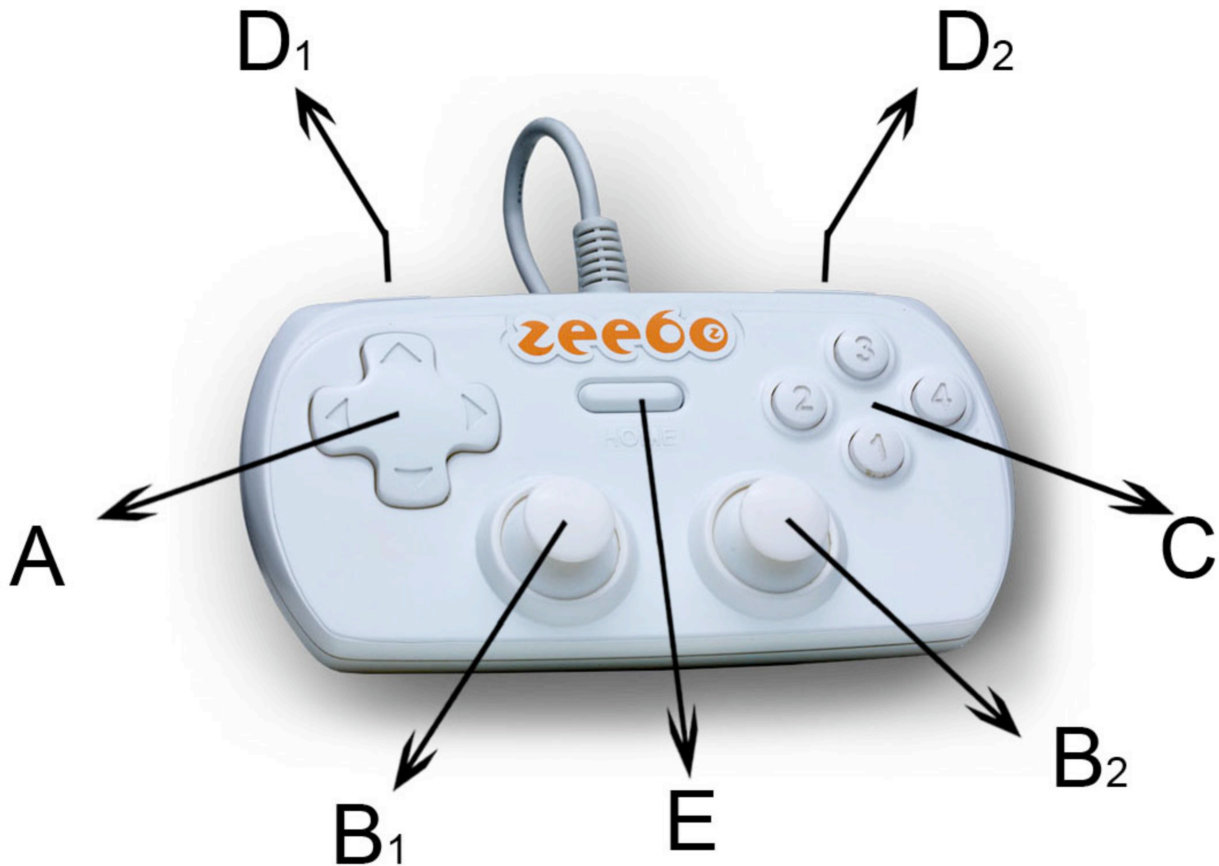


Figure 2-1 The Zeebo gamepad

A Directional pad:

Main buttons to navigate the UI and within the game menus. Moves are possible only in 4 directions up, down, right and left.

B1 and **B2** analog controls:

Both the right and left analog controls will be activated during the game when necessary. These controls provide more precise movement.

Only the left analog control is enabled for UI navigation following the same dynamic of the directional pad.

C Selection pad **1 – 2 – 3 - 4**:

1-2-3-4 buttons are mainly used during games featuring varied action and movement options such as shooting, jumping, accelerating, breaking, etc.

1 button will be used for the UI navigation as the enter button to confirm selections.

2 button will be used to go back to the immediate previous menu if the back button is not included in the screen.

3 button will be a sub-menu button that is informational and provides relevant details pertaining to the screen you are on. For example if the user is on the Game Data screen and presses the **3** button, the file size, game statistics, and game ranking information will be displayed. Informational details can be retrieved while on any screen by pressing the **3** button.

4 button will have different functionality according to each screen when needed. For example the **4** button can be used if an additional button is required for the third and fourth screens as a decision button.

D1 and **D2** trigger buttons (left and right):

Special buttons used in some games for accelerated weapon selection, car wheel control, etc.

E Home button:

The **Home** button redirects the user from any screen on the UI to the Stage. Every time the user presses this button, a confirmation screen will prompt the user to confirm the action.

For any menus inside your game, it is mandatory to follow the basic navigation guidelines seen in the Zeebo User Interface. The idea is to give a seamless experience to the consumer when navigating in the UI and playing a game.

2.3 Game File Size

Zeebo uses wireless carriers to deliver your game OTA to end users. Game file size may be a problem, due to airtime costs and also because some carriers might provide slow networks in some areas to deploy large files.

The file size of the game you're developing must match the commercial policy used by Zeebo Inc. in the target country.

The next sections describes the pricing categories that shall be followed by developers regarding file size

2.3.1 Low Price Category

Used for budget or casual titles. Game file size must be equal or lower than 8MB to be accepted by Zeebo Inc.

2.3.2 Standard Price Category

The category used for most titles available on the Zeebo deck. Game file size must be equal or lower than 25MB to be accepted by Zeebo Inc.

2.3.3 Premium Pricing Category

Please ask your commercial department to discuss the possibility of developing a game for Zeebo with more than 25MB of content.

Used in very rare cases where the developer needs a larger file size to provide the ultimate experience to a Zeebo user. Regardless, game file size must be equal or lower than 50MB to be accepted by Zeebo Inc.

2.4 Tools

2.4.1 QXEngine

QXEngine is a set of high level APIs for manipulating meshes and animations. A toolset is included for exporting and optimize content from 3D modeling and animation software. It is optimized for OpenGL ES on Qualcomm chipsets, providing content-side and engine-side optimzations. The main optimizations features are provided as libraries, which can easily be used in conjunction with existing engines.

The supported features includes:

- Meshes, exportable to Strips or Lists;
- Hierarchical and material animations;
- Shading: ambient, diffuse and specular;
- Blending: transparency and incandescence;
- Texture: all supported OpenGL ES formats, including compressed and uncompressed;
- Multiple cameras;
- Multiple culling options;
- A set of utility libraries, including Math, Data Structures, Command Manager, Diagnostics and Primitive Rendering;

A set of tools is also provided, including QStrip library, QXTextureConverter library and a Particle Systems Editor. More information about the QXEngine can be found on the documentation included with the installer.

QXEngine source-code is available upon signing an agreement with Qualcomm. If you want to develop your game for Zeebo using QXEngine, please contact devsupport@zeeboinc.com.

2.4.2 Adreno Profiler

The Adreno Profiler is an application for profiling OpenGL ES 3D based games. It provides a set of tools designed to detect bottlenecks and errors on the 3D graphics pipeline. Zeebo has an auxiliary processor that enables on hardware debug. More information about how to

use the Adreno Profiler can be found on the Adreno Profiler – Quick Start Guide, which is included with the installer.

3 Zeebo System Architecture

This chapter contains a description of the overall Zeebo system architecture. Sections on hardware and software architectures are provided, describing the capabilities that define the functionalities for the hardware and software present on the platform.

3.1 Hardware Architecture

The Zeebo wireless gaming platform uses the Qualcomm MSM7201A chipset, which is comprised of the following components:

- ARM11 processor @ 528Mhz – your application runs here. It also controls GPU and display.
- Adreno 130 Graphics Subsystem – provides the 2D and 3D feature set for graphics acceleration and functionality.
- DSP Application Subsystem – used to decode audio and implement some 3D features.
- MDP (Mobile Display Processor) – updates TV-Out from frame buffer.
- ARM9 processor – dedicated to modem duties.

These components are integrated on a 32 bit 133 Mhz bus.

Figure 3-1 below is a diagram of a high-level view of the Adreno 130 hardware architecture as it fits into the MSM7201A. The key subsystems shown are:

- ARM Subsystem with the ARM11 processor (applications processor) – main processor involved in control of the Adreno 130 graphics subsystem and overall display functionality.
- Adreno 130 Graphics Subsystem – provides the 2D and 3D feature set for graphics acceleration and hardware graphics functionality.
- SMI memory – a 32-bit-wide DDR stacked memory operating at 133 MHz internal to the MSM7201A, with 32MB of capacity.
- EBI memory on the AXI EBI-1 bus – a 16/32-bit-wide DDR memory operating at 133 MHz, with 128MB of capacity.
- IMEM – an internal SRAM consists of three banks of memory (128KB, 64KB, 64KB) accessible by the Adreno 130 graphics core.
- AMBA AXI buses – two 32-bit data buses, in the context of the graphics subsystem, mastered by the ARM11, and the Adreno 130 graphics core.

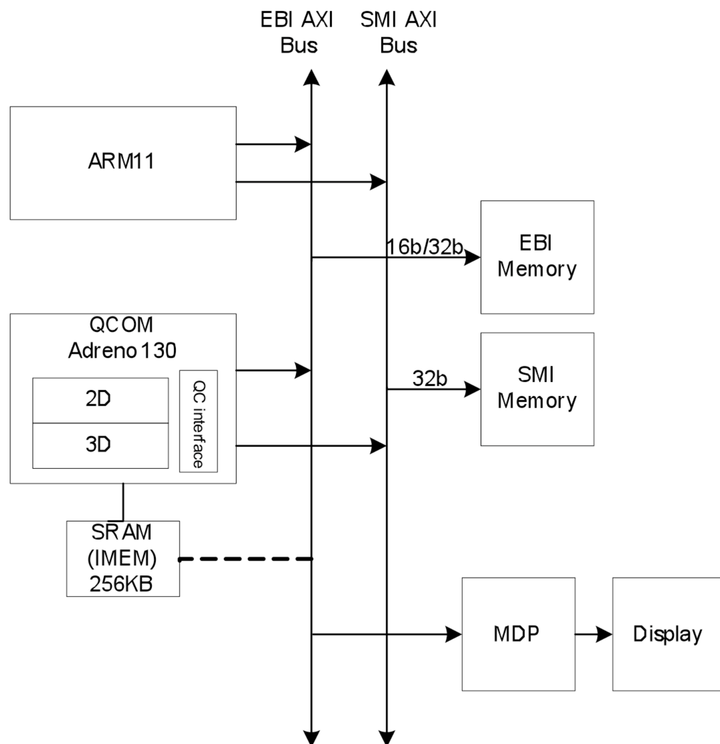


Figure 3-1 Graphics hardware block diagram

The Qualcomm Adreno 130 graphics core (GPU) has two AXI memory interfaces, one for EBI and one for SMI. Both of these interfaces are bus masters.

The ARM11 can send both commands and data to the GPU over the AXI buses via the EBI or SMI memory. Status information is transmitted over the AHB bus as well.

SMI and EBI memory are also used for display buffers (frame buffers). An image (2D or 3D) is built up in a specified buffer, and then the MDP transfers the buffer contents to the display when it is signaled from the GPU that a frame is ready.

For details of the memory usage, see chapter 5.

3.1.1 Memory Interface

The GPU has access to its local SRAM memory (which is external to the GPU core, but internal to the MSM7201A system) also called IMEM. This 4-part 64KB memory is broken into three logical sections of 128 KB, 64 KB, and 64 KB. This high-speed memory is used by the GPU for, most notably, 3D graphics operations as follows:

- Z-buffer;
- Pixel local buffers;
- Stencil buffer;
- Triangle bin buffer;
- Setup acceleration block (SAB) to pixel-pipe FIFO;

Zeebo Confidential and Proprietary

The GPU also accesses to the EBI and SMI memory banks, which is system memory. The current implementation allocates 2MB of SMI memory and 6MB of EBI memory for 3D graphics usage. This parallelism of memory access is designed for performance benefits.

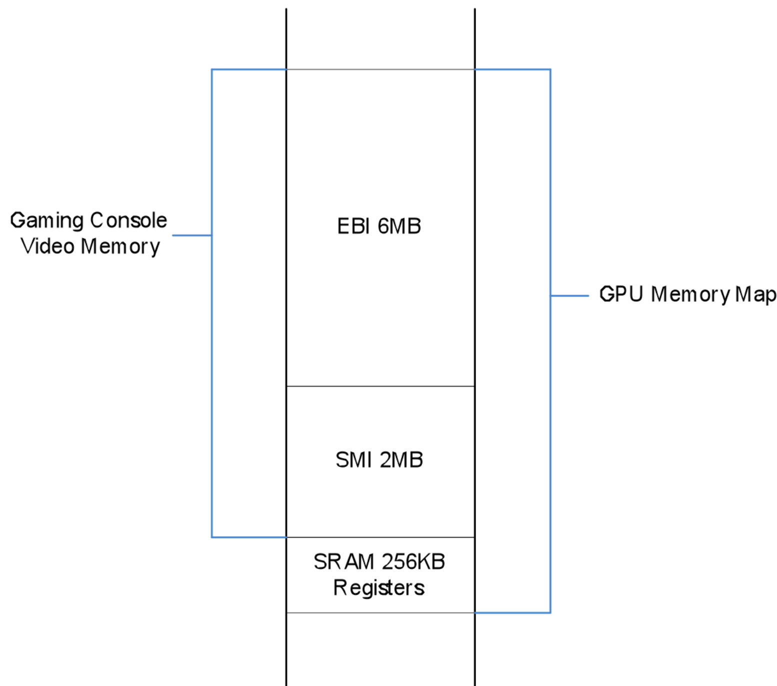


Figure 3-2 Video Memory layout

In BREW, there is a distinction between System Memory and Video memory, and the GPU has Memory Aperture into System Memory, making that memory region as Video Memory.

How BREW graphics software works:

- BREW places bitmaps into Video Memory as long as space exists.
- The display buffer(s) are located in Video Memory.
- The GPU accelerates BLTs where the source and destination are both located in Video Memory.
- BLTs are also accelerated between System Memory and Video Memory, but there is more than one copy involved.
- The more System Memory allocated to Video Memory, the more bitmaps can be stored there, thus improving performance.

3.1.2 Rendering Flow

The frame buffers (color buffers), Z-buffer and stencil buffers are located in the EBI memory. They will be copied to the SRAM for GPU to access for rendering. Once the rendering is done, it will be copied back to the buffers in EBI and updated to the LCD via MDP DMA engine directly (not going through the MDP internal buffers).

3.1.3 Binning

Since the 256KB of SRAM is not large enough to store all the different types of buffers, 'binning' mechanism was introduced. Binning renders subsections of the screen space to optimize memory usage.

Figure 3-3 illustrates the logical flow of how the rendering is done via the binning mechanism with QVGA frame buffers.

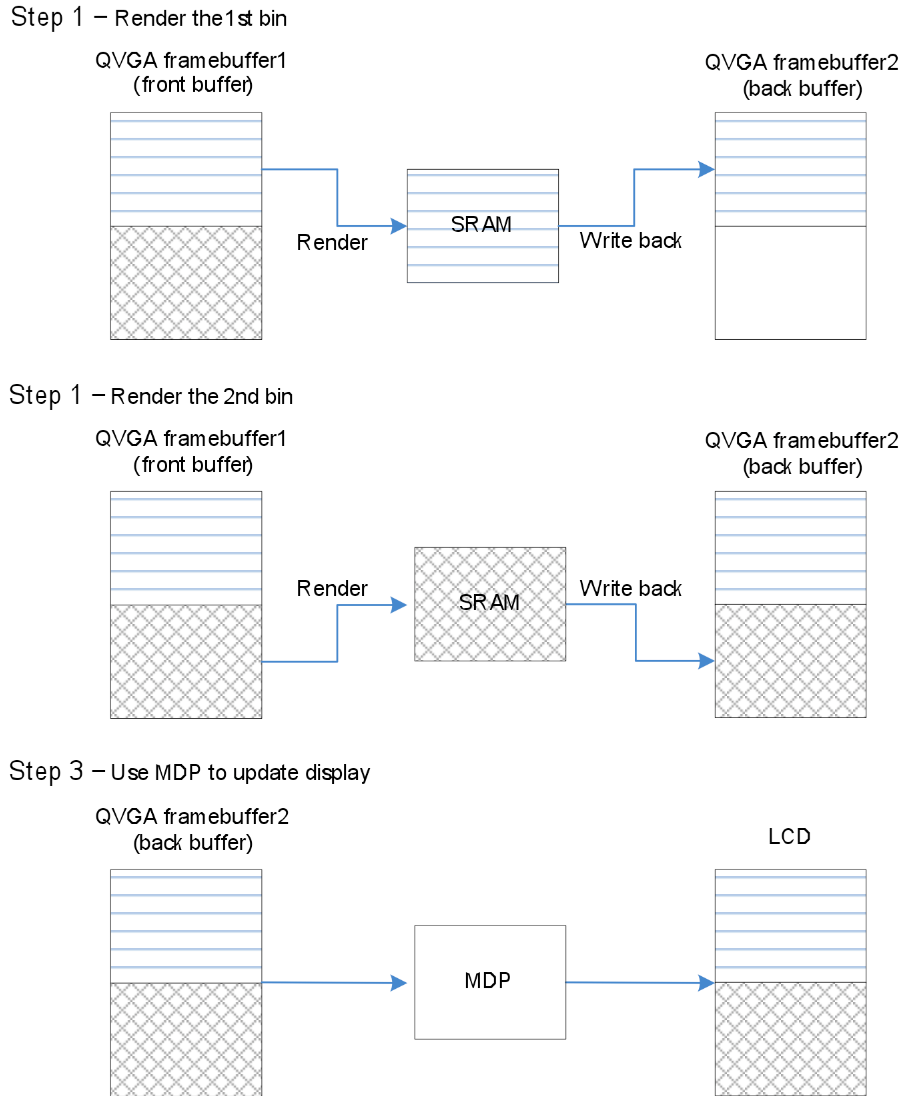


Figure 3-3 Binning with QVGA frame buffer

The number of bins depends on the frame buffer size. Presently, 2 bins for QVGA display, and 8 bins for VGA display, and 16 bins for WVGA display are used.

Zeebo will only support VGA (640x480) display configuration.

3.1.4 Ring Buffer

The ARM11 accesses the GPU via AHB interface for the transfer of status information and Ring Buffer read and write pointer synchronization. AHB interface is a slow bus interface. A Ring Buffer command mechanism is used for sending 2D/3D commands to the GPU through system memory (SMI or EBI).

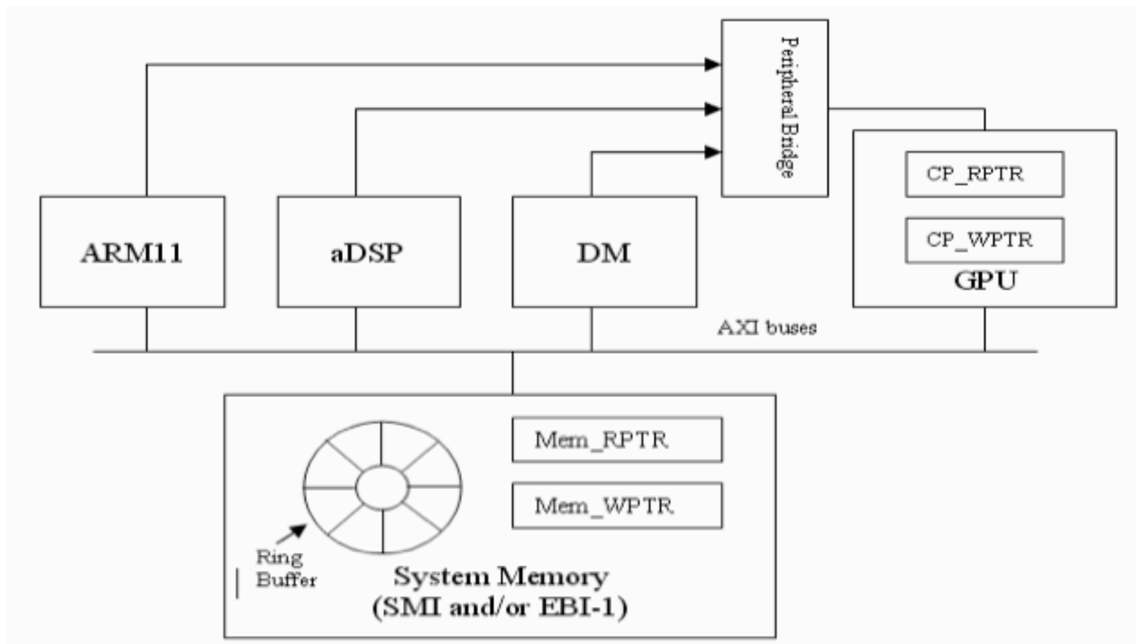


Figure 3-4 Layout of ring buffer commands for general operation

Figure 3-4 illustrates how the ARM11 places a command list (packet) into the Ring Buffer and updates the Mem_WPTR. Then the ARM11 writes CP_WPTR over the AHB bus to the GPU. The GPU then reads Mem_WPTR across the AXI bus and compares with CP_WPTR, then reads as appropriate the command stream from the ring buffer. Finally, the GPU updates the Mem_RPTR.

The command packets can contain indirect references to additional command lists located in other places (even in different memory). In this way, command lists can be reusable by simply placing the reference multiple times in a set of command packets.

3.1.5 Power Management

The GPU has its own internal logic for power management, being able to power down sections of the GPU when not in use. There is a GPU register that is used to enable this feature for each of the internal subsystems. The drivers have this feature enabled.

For the 3D core, the graphics driver turns on the power at 3D initialization routine and turns off at termination. In the actual implementation, `eglInitialize()` triggers the power on, and `eglTerminate()` triggers the power off via the Clock Regime power control

register for the GPU. The EGL function `eglInitialize()` is used to initialize the EGL display connection and `eglTerminate()` releases and terminates the EGL display connection.

For the 2D core, the current implementation is the display driver simply turns on the power upon device boot-up and does not turn off until the device shuts down. Improved 2D power management will be added in a later implementation to turn on and off the 2D core.

DCVS (Dynamic clock voltage system) is also used at a system level to affect the GPU power management.

3.1.6 Display Support

Zeebo provides VGA support (640x480) and it is possible to achieve more than 30 frames per second (fps) on the NTSC and PAL-M video systems used in most countries where the product will be sold. Refer to section 8.3 for more details on rescaling the video output.

PAL-M signals are identical to North American NTSC signals, except for the encoding of the color carrier. Therefore PAL-M will display in monochrome with sound on an NTSC set and vice versa. You should be fine using a NTSC TV for testing purposes. MDP processor will output signal in PAL-M as well.

Please dedicate a safe area for drawing – about 4-8 pixels each side (top and bottom). Older CRT TVs may distort the edges of the screen and text closer to each edge might disappear or be truncated. Refer to section 11.2 for more details on tuning your games for TV output.

3.2 Software Architecture

3.2.1 3D Graphics Architecture

The 3D Graphics standard OpenGL ES 1.0 Common Profile is supported on the Zeebo platform. This section explains the high-level architecture and its design.

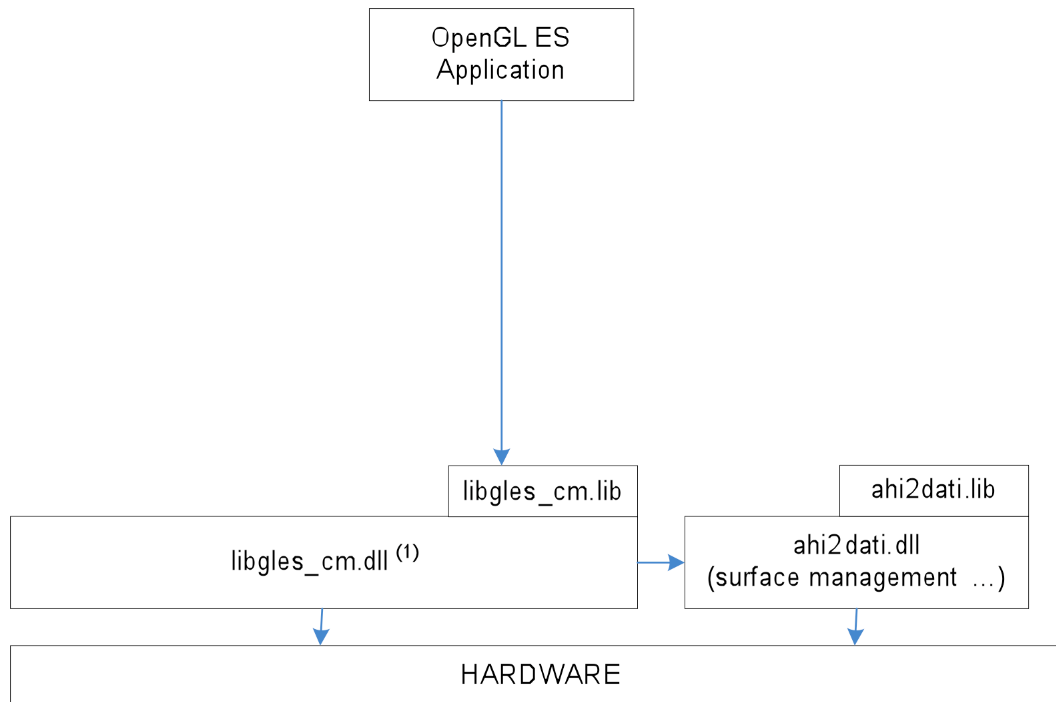


Figure 3-5 OpenGL ES 1.x Adreno 130 driver architectural overview

(1) Note that the early version of MSM7201A releases used glesati.dll instead of libgles_cm.dll.

Figure 3-5 illustrates how rendering calls are made logically. A BREW 3D graphic application makes calls through the OpenGL ES APIs.

The OpenGL ES driver (libgles_cm.dll) accesses the GPU for actual rendering. It also uses the AHI 2D interface (ahi2dati.dll) for basic surface management and issues commands to hardware.

3.2.2 Fixed-point Math Support

Floating point math is not hardware accelerated on the Zeebo wireless gaming platform, so for fast math operations it is strongly recommended to use fixed-point math. Below are some characteristics of the MSM7201A processor that might be useful for developers:

- ARM processor emulates floating-point math, which turns the operations slow;
- Integer math is fast but lacks fractional precision;
- GLfixed Format used is s15.16;
- Represented in a 32bit signed integer;
- Other choices are possible (for higher precision internal calculations);

Below are listed the major problems using fixed point math:

- Hard to interpret (when debugging). Some example values:

  i. 1 = 0x00010000

    ii. 2 = 0x00020000

    iii. ½= 0x00008000

    iv. Pi = 0x0003243F

- No standard math library available;

Qualcomm's Fixed Point Math Library is a C-style library (mathfixed.h and mathfixed.c), which is highly optimized for the ARM. It supports the following basic math functions:

- Type Conversions (float to fixed, int to fixed);
- add, sub, mul, div;
- sq, sqrt;
- log, pow;
- sin, cos, tan, asin, acos, atan, atan2;
- abs;

Note that there are no operators, for example:

x0 * x1 + y0 * y1 + z0 * z1;

turns into:

F_ADD(F_ADD(F_MUL(x0, x1), F_MUL(y0, y1)), F_MUL(z0, z1));

Float to Fixed conversion:

```
#define F2X(f) ((int32)((f)*65536.0f))

Fixed F2X(float f)
{
   //interpret our floating point as unsigned integer
   uint32 data = *(uint32*)&f;
   //extract the mantissa and add the leading 1
   uint32 mant = (data & 0x007fffff) | 0x00800000;
   //extract the exponent, de-normalize it (-127), shift by the
   //precision of the mantissa (-23) and multiply by the precision
   //of our fixed point (16)
   int8 exp = (int8)((data >> 23) & 0xff) -127 -23 + 16;
   //shift our mantissa by the remaining exponent
   mant = exp > 0 ? mant << exp : (exp < -23 ? 0x1 : mant >> (-exp));
   //apply sign bit
   return (data & 0x80000000) ? 0-mant : mant;
}

#define F_FACTOR 16
#define F_ADD(v1,v2) ((v1)+(v2))
#define F_SUB(v1,v2) ((v1)-(v2))
#define F_MUL(v1,v2) ((Fixed)((((int64)(v1)) * (v2)) >> F_FACTOR))
#define F_DIV(v1,v2) ((Fixed)(((int64)(v1) << F_FACTOR) / (v2)))
```

F_MUL (pseudo assembly):

Figure 3-6 F_MUL pseudo assembly

4 Operating Systems Overview

The Zeebo wireless gaming platform runs Qualcomm BREW 4.0.2. This section describes the basic services provided by the platform, including reference-counted objects, event handling, timers, and multitasking.

4.1 Creating and releasing BREW object instances

Your application interacts with BREW through instances of BREW objects created using the ISHELL_CreateInstance call. Creating an object instance is a time-consuming process, and as such, should be done as few times as possible.

For example, if you are loading 10 files, every file can use the same instance of the IFileMgr object to be opened. Instead of creating 10 instances of the IFileMgr, create one instance and use it 10 times to open the 10 individual files with the IFILEMGR_OpenFile() function.

Generally, you should try to create all your object instances when your application is first initialized. In some cases, you will need to create object instances in the middle of execution, but keep in mind that doing so will have a performance hit on graphics rendering, so try to minimize this.

After the object instances are no longer needed by your application, it is very important that they be released with the corresponding Release function. Not freeing an object before your application exits can cause unexpected behavior the next time your application is entered. Remember that the Release method of an interface is not the same as FREE() method, and calling FREE() on an object instance will crash the platform. The cleanup function specified in the call to AEEApplet_New() is called right before your application exits. It should free any remaining objects/memory that your application allocated during its execution.

It is generally not necessary to increase the reference count of an object with the AddRef() function every time you create an instance of an object. BREW will automatically increase the count of the object when it is created, and it will decrement it when you release it. Reference counting is used to keep track of the number of callers of an object.

4.2 Event Handling

There are some events that every application should handle. The EVT_APP_SUSPEND and EVT_APP_RESUME events are pair of events that should be handled by every app. An app receives the suspend event when it is about to be suspended. This could happen, for example, if the system firmware needs to display a message. This would cause the BREW application to suspend and the system firmware would take control of the interface. How

the suspend and resume events are handled is up to the application programmer. The easiest way is to return TRUE when the suspend event is received, and also return TRUE when the resume event is received. This will cause the app to start again from the beginning when the resume event is received.

Another event is the EVT_APP_NO_SLEEP event. When this event is received, BREW wants to make sure your application is still running. Returning TRUE means the application is still running. Returning FALSE means that your application did not handle this event, and BREW will permit the platform to go into low-power mode. At this point, you will see your application start to run really slowly. It's likely that every application will want to return TRUE to this event.

Returning the app to the state it was in before it was suspended requires more work. Every state (including OpenGL ES states) would have to be saved on suspend and then restored on resume. Also, when the suspend event is received, any pending actions should be discarded before handling suspend. For example, if the suspend event is received and there is an active timer event that is scheduled to expire, the timer should be cancelled before returning TRUE for the suspend event.

There are three main events related to keys on the phone: EVT_KEY, EVT_KEY_PRESS, and EVT_KEY_RELEASE. The EVT_KEY event is sent once anytime a key is pressed. This event should be used for keys that get pressed once and perform an action, i.e., toggling a state. The EVT_KEY_PRESS and EVT_KEY_RELEASE events are meant to work together for actions that require repeated action, i.e., holding down a key to move a player. Receiving EVT_KEY_PRESS indicates that a key is being held down, and the key is being pressed until an EVT_KEY_RELEASE is received by the app.

Pressing CLR (sometimes displayed as <<) to exit your app does not handle the AVK_CLR event, i.e., return FALSE. In this case, CLR will work just like END, and the app will be exited. If you want to use CLR for some other purpose, handle the AVK_CLR event and return TRUE in your event handler.

4.3 Cross Platform Programming

When developing your game, you will likely start by developing in the simulator (BREW SDK Extension for OpenGL ES 1.0 Common profile), and once you have something working, move to a Zeebo wireless gaming console. It is important to remember that because your game works in the simulator, it does not necessarily mean it will work the first time on target.

One of the most important things to remember when developing the game is that the program stack on the Zeebo is much smaller than the program stack you will get in the emulator. To avoid having stack overflows, do not place large amounts of data on the stack. For example, in almost all cases, your geometry and model data should be stored on the heap and never on the stack (even if this might work in the SDK). Stack overflows on the Zeebo will cause strange behavior and are difficult to debug. You should plan from the beginning to place large data on the heap.

Zeebo Confidential and Proprietary

Another common mistake is forgetting about byte ordering. The ARM processor on the Zeebo wireless gaming platform is big endian, as opposed to the little-endian processors on which the emulator runs. Thus, code that might compile and work in SDK may not for the Zeebo. For example, the code below initializes a buffer with the data "ABCD" and attempts to read this data into an integer. Option (1) will have different results read into the "value" variable if executed in the emulator and on the Zeebo, because the byte ordering is different and "unsigned ints" are in different orders. See section 5.2 for more detailed information about memory alignment issues on ARM processors.

However, by using MEMCPY as in option (2), the same data will be read into "value" because MEMCPY is implemented correctly on the respective platform.

```
unsigned int value;
char buffer[4] = “ABCD”;
/*(1)*/ value = *( (unsigned int*)&buffer[0] );//WRONG
/*(2)*/ MEMCPY(&value, &buffer[0], 4); //RIGHT
```

Along the same idea, keep in mind that you should be using MALLOC()/FREE(), which BREW provides instead of the standard malloc() and free(). Also, you should use the BREW standard library and string manipulation functions. When running on Zeebo, the C standard library is not available to your application.

Because dynamic modules have no read-write segment, there can be no static or global variables in a BREW application. While you can declare these when coding for the SDK; it will not compile or run on the target. This also limits the use of the C++ static keyword; you cannot declare class variables for the same reason.

The most common way to work with this is to place this kind of data in the heap and pass references to the data; one common place for this data is the application context variable BREW passes to your event handler.

While the most recent versions of both the GNU tool chain for Qualcomm BREW and Qualcomm's packaging utility elf2mod support some use of static and global variables through the introduction of fix-up code, it's strongly encouraged that you simply refrain from using them in the first place and place this data in the heap.

4.4 Timers

BREW provides millisecond-resolution timers that notify your application through a callback when a timer has expired. You can use this call your application's render loop after a timeout, control the frame rate of your application, and so forth. Timer values as small as 0 or 1 millisecond may be specified.

If you need to schedule an application to occur at the next available pass through the event handler and timing is not critical, use BREW's callback mechanism by specifying an AEECallback. Callbacks consist of a function pointer and context data, and are scheduled using ISHELL_Resume. Because ISHELL_Resume takes a reference to your AEECallback structure, it must be on the heap and not the stack.

4.5 Multitasking

The platform consists of several threads of execution in parallel, of which your application is but one thread. On BREW, multitasking is inherently cooperative: your application must periodically yield the processor so that other things can run. As a result, it's imperative that your event handler and other logic process events or perform actions quickly and then exit, giving other parts of the system adequate opportunity to run.

If you need to perform a long computation or loop, you have two choices. You can structure your code to operate asynchronously using callbacks, or you can implement the logic as the main method of an IThread instance. Under no circumstances should you use long loops in your code, because if you take too long to respond to an event or callback invocation, the platform will assume that your application has crashed and will reset.

Using callbacks lets you structure your code as a state machine; you break your code in to discrete states, and chain them using the callback mechanism described in Section 4.4. Each function assigned to the callback performs the work of a single state in your state machine; state transitions occur by scheduling the next state by initializing your callback and scheduling the operation using ISHELL_Resume.

This asynchronous callback approach is used by all of the BREW APIs that would traditionally block the processor on other platforms. For example, I/O is performed using the callback mechanism; you register a callback that BREW invokes when data is available to read. Similarly, you schedule a callback that the system will invoke when you can write data.

The IThread interface provides a software implementation of cooperatively scheduled threads of execution that you can use instead of the callback approach. (In fact, under the hood, IThread is implemented using BREW's callback mechanism.) Using IThread, you can structure your code as if it were executed synchronously, including letting you execute long loops such a a game's main thread. When using IThread, though, you must still yield the processor periodically by obtaining a callback to the current execution point and invoking ITHREAD_Suspend; you can see an illustration of this in the AEETU_Yield.c utility provided with the BREW SDK. The IThread interface provides you with a good abstraction when structuring control loops in your application.

4.6 Debugging

The BREW Simulator can be used for debugging games on the PC. Used in conjunction with Visual Studio, it is possible to set break points and place watches for variables used within your game. Output from your code can also be traced using the Output Window. Figure 4-1 illustrates the BREW Simulator configured to use the Zeebo device pack.

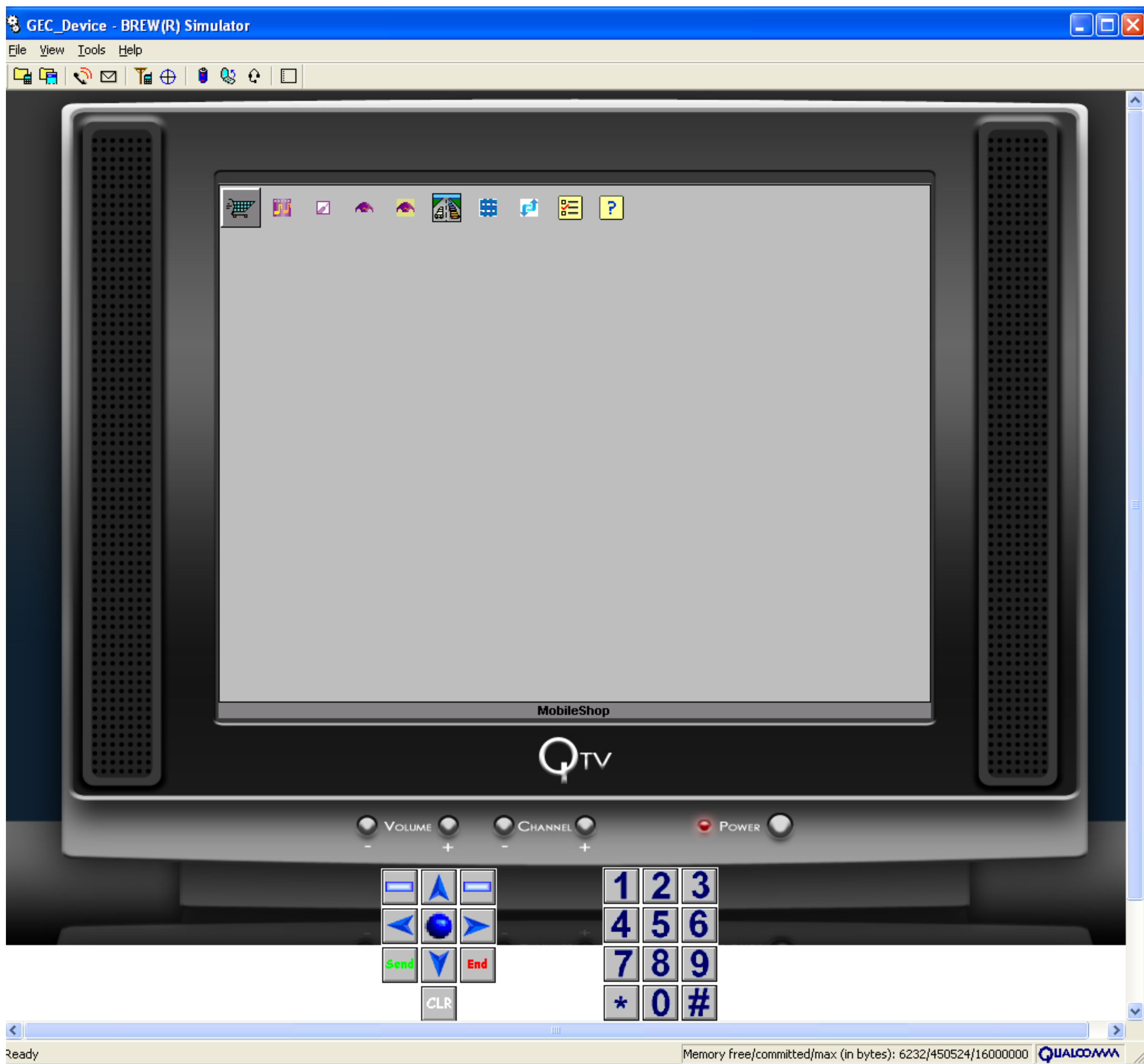


Figure 4-1 The BREW Simulator with Zeebo configuration

The BREW Debugger allows game developers to perform on-target debugging of Zeebo games using the GDB (GNU Debugger) or the AXD (ARMs Extended Debugger) and a USB cable connected to the device. The BREW Debugger supports standard debug operations such as:

- Starting and stopping execution of the application;
- Stepping through the code line by line;
- Examining and modifying variables;

Zeebo Confidential and Proprietary

5 Basic Memory Management

This section is intended to provide information concerning memory management and how to address issues related to address alignment on ARM processors. It also describes the total amount of memory that can be used to develop games for the Zeebo wireless gaming platform.

5.1 File System, Heap and Stack Sizes

The total file system size available on Zeebo is 1GB. This amount of memory is shared for all games preloaded or installed over the air. Be sure to check the free memory available before trying to save your game data files. Zeebo requires the usage of at most 64KB for save game data.

Heap size is limited to 32MB. Remember that less than 32MB will be available for you game. Some applications will be running at the same time when you start your game, and also remind that your binary file will be fully loaded into the heap before it starts its execution. So if your binary game takes up 2 MB, less than 30MB of heap will be available for loading game resources into the heap.

The Zeebo system has a 32KB of stack memory. Make sure your code doesn't allocate large array or structure variables locally; also try to avoid writing functions with a large scope. You can also use the IThread interface to let you cooperatively execute your code on a thread with a stack set in the heap. Using IThread you can set the total amount of stack you wish to use. Note that IThread provides only cooperative multitasking.

5.2 Memory Alignment Issues on ARM Processors

Memory accesses can be either aligned or unaligned. Aligned memory accesses occur when data is located on its natural size boundary. If the size of the data type is 4 bytes, for example, then it falls on its natural size boundary if it is located at a memory address that is evenly divisible by 4. Unaligned memory accesses occur in all other cases (in the example above, whenever the memory address is not divisible by 4).

ARM processors are designed to efficiently access aligned data. Attempting to access unaligned data on an ARM processor will result in either the incorrect data or significant performance penalties (these different symptoms will be discussed shortly). This contrasts with most CISC type processors (i.e., x86) in which access to 'unaligned data' is harmless.

5.2.1 Symptoms

The problem described above applies to all ARM architectures. However, depending on the availability of an MMU and operating system support, applications may see different behavior across different platforms. By default, unaligned memory accesses will not be trapped, and will result in the incorrect data. On platforms with an enabled MMU, however, the OS may trap the unaligned access and correct it at runtime. The result will be the correct data, but at a cost of 10-20 CPU cycles.

5.2.2 Common Causes

5.2.2.1 Type Casting

The following code illustrates a type casting pitfall:

```
void my_func(char *a)
{
   int *b = (int *)a;
   DBGPRINTF("%d", *b);
}
```

This simple example may result in an unaligned memory access, since we cannot guarantee the alignment of char *a is on a 4-byte boundary. This type of cast should be avoided whenever possible.

5.2.2.2 Working with Data Buffers

The most frequent cause of unaligned memory access stems from incorrect handling of data buffers. These data buffers might contain anything – data read from the USB port, over the network, or from a file. It is common for this data to be packed, meaning there is no padding inserted to ensure that data within the buffer lies on its natural size boundary. In this example, we will consider the case of loading a Windows BMP from a file and parsing the header.

A Windows BMP file consists of a header followed by the pixel data. The header is made up of two structures:

```
typedef PACKED struct
{
   unsigned short int type;          /* Magic identifier            */
   unsigned int size;                /* File size in bytes          */
   unsigned short int reserved1, reserved2;
   unsigned int offset;              /* Offset to image data, bytes */
} HEADER;

typedef PACKED struct
{
   unsigned int size;                /* Header size in bytes      */
```

```
   int width,height;                /* Width and height of image */
   unsigned short int planes;       /* Number of colour planes   */
   unsigned short int bits;         /* Bits per pixel            */
   unsigned int compression;        /* Compression type          */
   unsigned int imagesize;          /* Image size in bytes       */
   int xresolution,yresolution;     /* Pixels per meter          */
   unsigned int ncolours;           /* Number of colours         */
   unsigned int importantcolours;   /* Important colours         */
} INFOHEADER;
```

Note that the sizes of the HEADER and INFOHEADER structs are 14 and 40 bytes, respectively.

Lets assume that we want to determine the width and height of the image at runtime. The code to access this data might look like this:

```
#define INFOHEADER_OFFSET (sizeof(HEADER))
#define WIDTH_OFFSET (INFOHEADER_OFFSET + offsetof(INFOHEADER, width))
#define HEIGHT_OFFSET (INFOHEADER_OFFSET + offsetof(INFOHEADER, height))

int imageWidth, imageHeight;
void * fileBuf;

pMe->mFile = IFILEMGR_OpenFile(pMe->mFileMgr, "test.bmp", _OFM_READ);

if (pMe->mFile)
{
   IFILE_GetInfo(pMe->mFile, &fileInfo);
   fileBuf = MALLOC(fileInfo.dwSize);

   if (fileBuf)
   {
      result = IFILE_Read(pMe->mFile, fileBuf, fileInfo.dwSize);

      if (result == fileInfo.dwSize)
      {
         imageWidth = *((uint32*)(((byte*)fileBuf) + WIDTH_OFFSET));
         imageHeight = *((uint32*)(((byte*)fileBuf) + HEIGHT_OFFSET));
      }
   }
}
```

Note the offsets of the width and height. Because they fall on a half-word boundary, access to these values in the manner above will result in an unaligned memory access. Some of the recommended ways to avoid this problem are outlined below.

5.2.3 Recommended Solutions

5.2.3.1 Using MEMCPY

Our first option is to simply perform a MEMCPY() of the data from the buffer to our local variable:

```
if (result == fileInfo.dwSize) {
   MEMCPY(&imageWidth,
          (((byte*)fileBuf)+WIDTH_OFFSET),
          sizeof(uint32));

   MEMCPY(&imageHeight,
          (((byte*)fileBuf)+HEIGHT_OFFSET),
          sizeof(uint32));
}
```

The result is that the memory is copied byte-by-byte, avoiding any questions of alignment.

5.2.3.2 Using the PACKED compiler directive

Alternatively, we can use the PACKED compiler directive to allow use of pointers directly to the data we want, while forcing the compiler to handle the alignment issues. In the BREW environment, PACKED is defined as follows:

```
#ifdef __ARMCC_VERSION
  #define PACKED __packed
#else
  #define PACKED
#endif
```

By designating a pointer as PACKED, the ARM compiler will always generate the appropriate instructions to access the memory correctly, regardless of alignment. A modified version of the example above, using PACKED pointers, is given below:

```
#define INFOHEADER_OFFSET (sizeof(HEADER))
#define WIDTH_OFFSET (INFOHEADER_OFFSET + offsetof(INFOHEADER, width))
#define HEIGHT_OFFSET (INFOHEADER_OFFSET + offsetof(INFOHEADER, height))

PACKED uint32 * pImageWidth;
PACKED uint32 * pImageHeight;
uint32 imageWidth, imageHeight;
void * fileBuf;

pMe->mFile = IFILEMGR_OpenFile(pMe->mFileMgr, "test.bmp", _OFM_READ);

if (pMe->mFile)
{
   IFILE_GetInfo(pMe->mFile, &fileInfo);
   fileBuf = MALLOC(fileInfo.dwSize);

   if (fileBuf)
   {
      result = IFILE_Read(pMe->mFile, fileBuf, fileInfo.dwSize);

      if (result == fileInfo.dwSize)
      {
         pImageWidth = (uint32*)(((byte*)fileBuf) + WIDTH_OFFSET);
         pImageHeight = (uint32*)(((byte*)fileBuf) + HEIGHT_OFFSET);
         imageWidth = *pImageWidth;
         imageHeight = *pImageHeight;
```

```
      }
   }
}
```

5.2.3.3 Defining well-aligned data structures

While programmers will typically have no control over standardized data formats, such as the BMP header used in the example above, when defining your own data structures you should be sure to lay out the data in a well-aligned way. The following basic example demonstrates this principle:

```
#ifdef __ARMCC_VERSION
typedef PACKED struct {
   short a;       // offsetof(a) = 0
   int   b;       // offsetof(b) = 2 – misalignment problem!
   short c;       // offsetof(c) = 6
} BAD_STRUCT;

typedef struct {
   int   b;       // offsetof(b) = 0 – no problem!
   short a;       // offsetof(a) = 4
   short c;       // offsetof(c) = 6
} GOOD_STRUCT;
```

Simply by rearranging the order in which we declare the struct members, we can resolve some of the alignment issues. Also note that if BAD_STRUCT is not declared as PACKED, the compiler will typically insert padding such that each field is well aligned. This, however, is usually undesirable as it wastes memory and can almost always be avoided simply by declaring fields in order of decreasing size.

5.2.4 Testing With the BREW Simulator

BREW Simulator versions 3.1.2 and above provide the ability to turn on data-alignment checking. When this feature is enabled, the BREW Simulator will display a dialog informing you of each unaligned memory access and provide you with the option of ignoring the problem or breaking into the code.

Refer to the BREW SDK User Docs section titled “Misaligned Data Exception Support” for further information on this feature.

Note: Because the x86 architecture does not have any issues with accessing unaligned data, you cannot compile the Simulator DLL using the __packed directive (which is why PACKED is defined as whitespace in the WIN32 environment). This means that unaligned accesses that are resolved by using PACKED pointers will still trigger the Simulator’s alignment check.

6 Input/Output

6.1 Understanding the Zeebo Gamepads

Zeebo gamepads are designed for a high-quality gamming experience. Figure 2-1 illustrates the gamepad.

- A: 1 directional pad - used to navigate on the UI and within the game menus.
- B: 2 analog controls - used during the game when necessary, providing more precise movement.
- C: 4 game buttons in the top of the control – mainly used during games featuring varied action and movement options such as shooting, jumping, accelerating, breaking, etc.
- D: 2 trigger buttons (left and right) - special buttons used in some games for accelerated weapon selection, car wheel control, etc. These buttons can also be used while navigating the UI to quickly cycle backwards or forwards to previously navigated screens (similar to Internet Explorer's browser buttons).
- E: 1 home button – when pressed for more than 3 seconds, will redirect the user to a confirmation screen, asking if he wants to go back to the Stage and exit the game, or return to the game he was playing. Just pressing and releasing it will act as a pause button for the game.

Development can also be done with Logitech Dual Action gamepads. Please do not consider the second layer of shoulder buttons while developing your game.

6.2 IHID Overview

The IHID and IHIDDevice interfaces provide access to USB Human Interface Device (HID) keyboards, mice, and gamepads that are attached to the device. The IHID interface provides information about which devices are attached as well as notification about device insertion/removal. The IHIDDevice interface provides access to the state of a particular device.

6.2.1 Using ISignal

The ISignal interface serves asynchronous notifications to the application. It works similarly to an AEECallback, except that it is an actual object as opposed to just a structure. You create ISignal objects using the ISignalCBFactory interface.

Zeebo Confidential and Proprietary

The notifications in the IHID and IHIDDevice interfaces rely on ISignal objects. ISignal provides similar functionality to AEECallbacks but is created and used slightly differently. The first thing that is needed is to create the ISignalCBFactory.

```
ISignalCBFactory *piSignalFactory;
int nErr;

nErr = ISHELL_CreateInstance(pIShell,
     AEECLSID_SignalCBFactory,
                             (void**)&piSignalFactory);
```

You can then use this object to create an ISignalCtl object:

```
nErr = ISignalCBFactory_CreateSignal(piSignalFactory,
                                     SignalCallbackFunction,
                                     pUser,
                                     (ISignal**)0,
                                     &piSignalCtl);
```

The ISignalCtl can then be cast to an ISignal for functions that require it and SignalCallbackFunction(pUser) will be invoked when the signal is set. When you are done with the ISignal you should call ISignalCtl_Detach before releasing the object to prevent your callback from being invoked.

6.2.2 Using IHID to determine which devices are attached

The IHID interface provides methods for determining which devices are attached at any given moment and for receiving notifications when a device is inserted or removed.

In order to use this interface you must first create it:

```
IHID *piHID;
int   nErr;

nErr = ISHELL_CreateInstance(pIShell, AEECLSID_HID,
                             (void**)& pIHID);
```

This object can then be used to determine what devices are attached:

```
int nConnectedDevices;
int *pDevHandles;
int nErr;

nErr = IHID_GetConnectedDevices(pIHID,
                               AEEUID_HID_Joystick_Device,
                                NULL,
                                0,
                                &nConnectedDevices);
if((SUCCESS == nErr) && (nConnectedDevices != 0))
{
   pDevHandles = (int *)MALLOC(nConnectDevices * sizeof(int));
   if(NULL != pDevHandles)
```

```
    {
        nErr = IHID_GetConnectedDevices(pIHID,
                                        AEEUID_HID_Joystick_Device,
                                        pDevHandled,
                                        nConnectedDevices,
                                        &nConnectedDevices);
        if(SUCCESS == nErr)
        {
            int nIndex;
            for(nIndex = 0; nIndex < nConnectedDevices; nIndex++)
            {
                //Device with handle of pDevHandles[nIndex] is present
            }
        }
    }
}
```

This will give you the list of devices that are present when the code is executed. If you are interested in detected when devices are inserted or removed you must register a signal with the IHID interface.

```
IHID_RegisterForConnectEvents(pIHID, (ISignal *)piDeviceSignal);
```

In the handler for this signal you will need to get all of the connected events

```
boolean bEventsDropped;
int nHandle, nStatus;

while(AEE_SUCCESS == IHID_GetNextConnectEvent(pIHID, &nHandle,
                                              &nStatus,
                                              &bEventsDropped))
{
    AEEHIDDeviceInfo di;

    if(bEventsDropped)
    {
        break;
    }

    //Check to see if it a joystick
    if(SUCCESS == IHID_GetDeviceInfo(pMe->m_pIHID, nHandle, &di))
    {
        if(di.nDeviceType == AEEUID_HID_Joystick_Device)
        {
            if(AEE_SUCCESS == nStatus)
            {
                NewJoystick(nHandle);
            }
            else
            {
                DeleteJoystick(nHandle);
            }
        }
    }
}
```

```
if(bEventsDropped)
{
    //The system dropped a connect/disconnect event
    //Flush the queue
    while(AEE_SUCCESS == IHID_GetNextConnectEvent(pIHID, NULL,
                                                  NULL,
                                                  NULL))
    {
       //Do Nothing
    }
    //The user should now reread the list of connected devices using
    //IHID_GetConnectedDevices and update its internal state.
}
```

6.3 Understanding System I/O

The IHIDDevice provides information about a specific device connected to Zeebo. These devices are presented as a set of axis and buttons. The API has no limitations concerning the number of buttons that are supported. The axes are limited to X, Y, Z, rX, rY, rZ, vX, vY, vZ, aX, aY, aZ, fX, fY, fZ.and they can be either absolute or relative.

Games running on Zeebo can either poll or register for notification for button or axes state change information. The events generated by the buttons are captured in an event queue. For each axes and button there is a minimum value, maximum value and a unique ID associated.

The following sections describe how to use the IHIDDevice interface for handling gamepad events and state change information.

6.3.1 Creating a IHIDDevice reference

Once the device handle of the required device has been identified through the IHID interface, the IHIDDevice object for this device can be created with IHID_CreateDevice.

6.3.2 Button Events

After creating an IHIDDevice object using the handle for the required device, attach a signal handler using IHIDDevice_RegisterForButtonEvent to be notified for button press/release events.

Once this event is received button information can be retrieved using IHIDDevice_GetNextButtonEvent. The AEEHIDButtonInfo struct retrieved by this call would have the required information about the button. The nButtonUID member will be 0 if the UID is unknown.

6.3.3 Axis Events

After creating an IHIDDevice object using a handle for the required device type, you will need to attach a signal handler using IHIDDevice_RegisterForPositionChange to be notified

for axis change events. Once the signal is received, axis information can be retrieved by calling IHIDDevice_GetPositionState. The AEEHIDPositionInfo struct returned by this call would have the axis information. If bRelativeAxes member is set to true the axis information reported from the device is relative.

6.3.4 Device Events

All the device status query and event handling functions accessible through IHID API using the device handle are accessible using the IHIDDevice API. This provides an optimal way for filtering out the device status events once a corresponding IHIDDevice reference is available.

6.3.5 Gamepad Rumble

If IHIDDevice_GetRumble() is successful the current rumble status will be retrieved. If the device does not support rumble then this function will return AEE_EUNSUPPORTED.

IHIDDevice_Rumble() sets the rumble state as requested. The input values should be in the range of 0 to 65,535. If the device does not support rumble then this function will return AEE_EUNSUPPORTED.

```
IHIDDevice_Rumble(*po, 0xFFFF, 0);
```

The above snippet sets the left motor intensity to max and right motor intensity to 0.

Remember that current version of Zeebo gamepad does not support rumble.

6.3.6 Exclusive Access

The priority level for an application using this device can be set with IHIDDevice_SetExclusiveLevel(). The default value for this level is 0, so all the applications accessing a particular device will be notified of device changes. The default BREW events that are sent for a given device will only be sent if no instances of IHIDDevice for that device have a non-zero exclusive level.

If this level is updated to a positive integer value greater than 0, all the instances of this device which have a level lower than this value will not receive any notifications.

Eg:

Consider 3 applications accessing a keyboard:

- App 1 – Word Processing – 0
- App2 – Keyboard test application - 0
- App3 – Keyboard controlled game – 0

In the above case the Keyboard notifications are delivered to all the apps. Now, if the priority of the device from App 3 is changed from 0 to 1, the notifications will only be received by App3.

Setting this parameter from one application to create parity in exclusivity would result in reduction of BREW events generated.

6.3.7 Default Event Handling

Regular BREW events will also be sent to the attached HID devices. When a keyboard is present, it will send EVT_KEY events or EVT_CHAR events depending on which key is pressed. Mouse events will use similar events as touch screen.

Gamepads events are based on the UID of the key. Bellow is the default event mapping for the gamepad:

- AEEUID_HIDJoystick_DPad_Up - AVK_UP
- AEEUID_HIDJoystick_DPad_Left - AVK_LEFT
- AEEUID_HIDJoystick_DPad_Down - AVK_DOWN
- AEEUID_HIDJoystick_DPad_Right - AVK_RIGHT
- AEEUID_HIDJoystick_Start - AVK_SELECT
- AEEUID_HIDJoystick_Back - AVK_CLR
- AEEUID_HIDJoystick_Left_Thumbstick - AVK_SOFT1
- AEEUID_HIDJoystick_Right_Thumbstick - AVK_SOFT2
- AEEUID_HIDJoystick_Button_1 - AVK_GP_1
- AEEUID_HIDJoystick_Button_2 - AVK_GP_2
- AEEUID_HIDJoystick_Button_3 - AVK_GP_3
- AEEUID_HIDJoystick_Button_4 - AVK_GP_4
- AEEUID_HIDJoystick_Left_Shoulder_Upper - AVK_GP_5
- AEEUID_HIDJoystick_Right_Shoulder_Upper - AVK_GP_6
- AEEUID_HIDJoystick_Left_Shoulder_Lower - AVK_GP_SL
- AEEUID_HIDJoystick_Right_Shoulder_Lower - AVK_GP_SR

6.4 Zeebo Gamepad Remapping

In order to have the correct mapping of joystick buttons and analog sticks from Zeebo gamepad, a set of helper functions are provided along with the SDK. Refer to AEEHIDThumbsticks.c and AEEHIDButtons.c located inside your BREW SDK src folder for further details on each helper function provided.

Zeebo Confidential and Proprietary

The button remapping is done replacing the IHIDDevice_GetNextButtonEvent() function call with AEEHIDButton_GetNextButtonEvent() provided in AEEHIDButtons.c file.

Analog stick remapping is achieved replacing IHIDDevice_GetPositionState() function call with AEEHIDThumbstick_GetPositionState() call, declared in AEEHIDThumbsticks.c file. Note that developers must initialize the analog stick remapping every time a new joystick is connected with AEEHIDThumbstick_InitializeMapping() before calling AEEHIDThumbstick_GetPositionState(). Also you must call AEEHIDThumbstick_DestroyMapping() every time a joystick is removed in order to destroy the mapping.

The Home button event generated while pressing/releasing it is AVK_CLR, and pressing/releasing the right thumbstick will generate an AVK_SELECT. The AVK_SELECT key event is used for launching games while in the Qualcomm UI (application list).

7 3D Graphics

Zeebo supports the OpenGL ES 1.0+ Common and Common-Lite profiles. The major features provided by OpenGL ES 1.1 standard are provided in the GL and EGL extensions present in the hardware.

This chapter highlights the key features supported by the Zeebo 3D rendering system.

7.1 OpenGL ES Overview

7.1.1 Frame Buffer (Color)

The frame buffer uses a 16-bit 565 RGB format, as this most closely represents the native format of the video output. A 16-bit depth buffer and 4-bit stencil buffers are optional.

7.1.2 Color Buffer Extension

This Color Buffer extension is deprecated. Limited support is still available however we strongly suggest using regular OpenGL functions since there is no native 2D/3D synchronization support on the MSM7201A. Blitting textures to the screen for overlay effects can be accomplished using the glDrawTex__ extension.

For further portability, applications should take into account the fact that this extension may not be available on future releases of the platform.

7.1.3 Textures

Zeebo supports texture sizes of up to 1024 x 1024. Texture dimensions must be a power of 2. At the API level, Zeebo handles all texture formats defined by the OpenGL ES specification.

Zeebo includes an 8KB texture cache. Optimal performance can be attained when an entire texture can fit in the cache, e.g., 64 x 64 x 16bpp, 32 x 128. In general, texture images should be as small as possible without sacrificing quality. Texture compression is performed after the texture cache so a significant gain can be expected both in performance as well as in texture cache use. See the next section on texture compression for details.

Mipmapping is supported in hardware for Zeebo. The ideal filter settings for best quality and performance are GL_LINEAR_MIPMAP_NEAREST for minification and GL_LINEAR for magnification. Trilinear filtering is not supported in hardware. Setting the filter settings to GL_LINEAR_MIPMAP_LINEAR will result in a significant performance reduction. For

glCopyTexImage2D() and glCopyTexSubImage2D(), keeping the frame buffer and texture formats to 16-bit RGB 565 is desirable for performance.

There is no difference in performance between filtering environment modes (GL_REPLACE, GL_DECAL, GL_MODULATE, GL_BLEND). Also, there is no performance degradation with respect to smooth and flat shaded triangles.

Zeebo supports two texture units as well as the texture crossbar extension allowing each texture combiner unit to accept input from both textures. Multitexturing effects in a single pass can significantly enhance the visuals of a game without incurring much of a performance penalty

When creating a texture (using glTexImage2D(), glCopyTexImage2D() or glCompressedTexImage2D()), OpenGL ES creates a copy of the texture data, as required by the OpenGL specification. Therefore, the application can free its copy of the texture data. The texture data is not needed during the lifetime of this OpenGL context.

7.1.4 Back-face Culling

Back-face culling is the process by which polygons that are not facing the camera are removed from the rendering pipeline. This is done by comparing the polygon's surface normal with the position of the camera. Back-face culling can be considered part of the hidden surface removal process of a rendering pipeline. The usage of back-face culling is recommended to improve the performance on the rendering pipeline.

7.1.5 Fog

Fog coefficients are calculated at each vertex then interpolated during rasterization (per-vertex fog). Artifacts common to vertex-fog processing will appear such as vertices outside, i.e., in front of the fog layer being "covered" by fog.

7.1.6 Clearing the Color Buffer

If the game draws to the whole screen for each frame a little bit of time can be saved by not clearing the color buffer.

7.1.7 Batching

As with most graphics hardware, Zeebo performs best when working with large batches of data. There is some setup involved every time you call glDrawElements() or glDrawArrays(). When rendering only a few triangles at a time, the cost of the setup becomes the bottleneck. The ratio of the setup cost to the rendering cost is significantly reduced when drawing dozens of triangles at once, and optimal with hundreds of triangles.

Multiple groups of strips and fans should be concatenated into a single group using degenerate triangles. Individual triangles as well as really small strips and fans (typically consisting of less then 3 triangles) can be grouped together into a single triangle list.

Zeebo Confidential and Proprietary

7.1.8 Stencil

Zeebo supports a 4-bit stencil buffer.

7.1.9 Blending

Blending can be enabled without any significant performance degradation.

7.1.10 Lightning

There is no hardware support for OpenGL ES lighting. Try to use multitexture lightmapping to light static geometry. Minimize dynamic lighting to only the moving objects in the game and try to balance triangle count of these objects between quality and performance. When using lighting try to use a single directional diffuse light for best performance.

7.1.11 Data Types and Precision

The OpenGL ES implementation on Zeebo comes in two profiles. OpenGL ES Common profile allows the use of floating point data types while OpenGL ES Common-Lite only use byte, short and fixed data types. It is important to remember that most embedded platforms (especially the ARM based platforms) do not have native floating-point support available. In some cases floating point is emulated in software which can cause severe performance issues for intense floating point calculations.

The OpenGL ES Common-Lite Profile for Zeebo allows you to specify geometry data as byte, short, or fixed-point data (GL_BYTE, GL_SHORT, GL_FIXED). The range of values each accepts respectively are:

- GL_BYTE – [-128,127]
- GL_SHORT – [-32768,32767]
- GL_FIXED – [-32768, 32767], with 16 bits of fractional precision

As opposed to floating-point systems where precision is not a major issue, with a fixed-point system, it is quite easy to exceed the acceptable threshold on the data if one is not careful. It is up to the application programmer to keep in mind the precision of the geometry data. For example, as indicated above, the range for GL_FIXED is -32768 to 32767, along with 16 bits of fractional precision. The programmer must ensure that matrix operations applied to the data will never cause an overflow. Suppose the current MODELVIEW matrix has a scaling factor of (128, 128, 128) and this is being applied to the point (2000, 2000, 2000). The result is (128 * 2000, 128 * 2000, 128 * 2000) = (256000, 256000, 256000), which is outside the range of values that can be handled by GL_FIXED. This will cause undesired rendering behavior. Again, keeping in mind that Zeebo is a fixed-point system is very important.

Zeebo Confidential and Proprietary

Also, try to preprocess your data as much as possible so that it does not have to be done at runtime. Just for reference, the instructions for doing fixed-point multiplication and division are shown here:

```
//multiplication
__int64 c=(__int64)a*b;
c = (__int64)c>>16;
//division
__int64 c = (__int64)a<<16;
c = (__int64) c/b;
```

7.1.12 Extended Data Types

To gain a memory usage savings it is possible to use GL_SHORT for texture coordinates. However this has the disadvantage that the texture view matrix needs to apply a scaling factor to the shorts. There is a small performance penalty for using this. Instead the application should check to see if the ATI_extended_texture_coordinate_data_formats extension is available and use one of the following formats which is supported by Zeebo hardware and don't need any scaling: GL_BYTE_4_4_ATI GL_SHORT_4_12_ATI GL_SHORT_8_8_ATI.

7.2 Supported OpenGL ES extensions

Table 7-1 summarizes the supported OpenGL ES 1.1 features that are hardware accelerated and are exposed using the EGL and GL extensions mechanism.

Table 7-1 OpenGL ES extensions support

Features	Extensions	Zeebo
Automatic Mipmap Generation	SGIS_generate_mipmap	No
Buffer Objects	ARB_vertex_buffer_object	Yes
Draw Texture	OES_draw_texture	Yes
Matrix Get	OES_matrix_get	No
Matrix Palette	OES_matrix_palette	Yes
Multitexture	-	Yes
Point Parameters	ARB_point_parameters	No
Point Size Array	OES_point_size_array	Yes
Point Sprites	OES_point_sprite	Yes
Rendering to Textures	-	No
State Queries	-	Some
Texture Combine	ARB_texture_env_combine	Yes
Texture Dot3	ARB_texture_env_dot3	Yes
User Clip Planes	-	No

7.3 Using OpenGL ES and EGL APIs in BREW

Since every BREW application is built as a dynamic downloadable application, each requires a set of interfaces to be created that provide dynamic binding at runtime. With that said, a set of BREW interfaces has been created that provide wrappers so the standard OpenGL ES and EGL APIs can be called.

Because there are multiple GL and EGL interfaces available we strongly recommend using the wrapper functions and call the standard OpenGL ES APIs directly. The wrapper functions will take care of querying and initializing the highest available GL and EGL interfaces.

Note: Using ISHELL_CreateInstance with parameters AEECLSID_GL or AEECLSID_EGL has been deprecated. If a user insists on using the BREW interfaces they should look at EGL_1x.c and GLES_1x.c.

7.3.1 Steps for using standard OpenGL ES API

- Include the IEGL and IGL BREW interface wrapper headers

```
#include <EGL_1x.h>
#include <GLES_1x.h>
```

- Include the standard EGL and GL include files

```
#include <gles/egl.h>
#include <gles/gl.h>
```

- Initialize the IEGL and IGL interfaces (in that order)

```
if (EGL_Init( pMe->a.m_pIShell ) != SUCCESS) { return FALSE; }
if (GLES_Init( pMe->a.m_pIShell ) != SUCCESS) { return FALSE; }
```

- Use regular EGL and GL functions like eglGetDisplay() and glClear()…
- Before exiting the application release the IGL and IEGL interfaces (in that order)

```
GLES_Release();
EGL_Release();
```

- Include the IEGL and IGL BREW interface implementation files EGL_1x.c, GLES_1x.c into your project.
- When using OpenGL ES Extensions also include <gles/glESext.h> and build your project with GLES_ext.c.

7.4 Supported 3D Graphics API

OpenGL ES 1.0 and most of OpenGL ES 1.1 API support both Common and Common Lite profiles. The Common profile is enabled by default; this includes OpenGL ES 1.0 plus most of the OpenGL ES 1.1 features with the exception of state queries. OpenGL ES 1.1 features are implemented as standard OpenGL ES Extensions, so it can be queried using the standard OpenGL ES and EGL query mechanism which functions are supported and which are not. A complete list of OpenGL ES functions supported are listed in Appendix B.

7.4.1 Summary of 3D Accelerated Rendering Support

- Color depth format 565 – Color Buffer at 16 bpp
- Double buffered Color Buffer
- Full resolution 16-bit Z-buffer
- Stencil Buffer at 4 bpp
- Programmable input vertex formats
- Indexed and embedded vertex lists
- Primitives – single triangle, strip, fan, point sprite
- Vertex transformation engine
- Trivial rejection capability
- Vertex clip check against frustum and user-defined clip planes (software performs clipping)
- View-port transform
- Back-face culling
- Polygon offset capability
- Vertex skinning support (with up to two matrices)
- Multi-texturing – dual texture filtering units at maximum 1024x1024 resolution
- Texture filtering for dual textures – nearest and bi-linear
- Texture filtering for single textures – nearest, bi-linear and tri-linear
- Texture formats – all OpenGL 1.0 formats
- Texture Compression
- Perspective correct texturing
- 32-bit precision in pixel-pipe
- Mip-mapping – 10 maps per texture with per-pixel level of detail (LOD)
- Scissoring
- Alpha test
- Dithering
- Logic operations
- Alpha blending with destination pixels
- Specular color blending
- Z and color buffer masking
- Z buffer depth testing with 16-bit Z
- Vertex fog interpolation
- Lighting supported in SW
- Per fragment Fog

Zeebo Confidential and Proprietary

- Point Sprites
- Dot 3 lighting

7.4.2 3D Graphics Limitations

Multiple EGL contexts are not supported by the graphics driver at this time. Therefore, only one 3D graphics application may run at a time; moreover, your application must share its EGL context throughout, and not create separate contexts.

8 3D Graphics Optimization and Tuning

This section describes the performance and other characteristics of the OpenGL ES implementation and provides recommendations for attaining optimal performance and results.

8.1 Working with Vertex Buffer Objects (VBO’s)

Vertex Buffer Objects allow static vertex data to reside in graphics memory and significantly reduce the bus bandwidth usage. This optimization is a must have for every OpenGL game running on the MSM7201A chipset.

Vertex Data needs to be transferred over the bus to graphics memory each time it is used. When dealing with large data sets it is possible that the bus bandwidth limitation becomes the bottleneck for the application. VBO’s allow the application to upload static data into graphics memory (similarly to textures) and leave it there for as long as it is needed.

For applications to get real acceleration, there are some restrictions:

- It is not accelerated for GL_LINE_LOOP: these fall back to traditional vertex packing routines.
- It is not accelerated when GL_LIGHTING is enabled or for matrix-palette skinning

8.1.1 Preparing the VBO function extensions

Sample Code:

```
char* pszExtensions = (char*)glGetString(GL_EXTENSIONS);

/*
* check to see if the vertex buffer extensions are available
*/
if (STRSTR(pszExtensions, "ARB_vertex_buffer_object") == NULL)
{
   return FALSE;
}
/*
* setup the function pointers
*/
pMe->glGenBuffers =
(PFNGLGENBUFFERSARBPROC)eglGetProcAddress("glGenBuffersARB");
pMe->glDeleteBuffers =
(PFNGLDELETEBUFFERSARBPROC)eglGetProcAddress("glDeleteBuffersARB")
pMe->glBindBuffer =
(PFNGLBINDBUFFERARBPROC)eglGetProcAddress("glBindBufferARB");
```

```
pMe->glBufferData =
(PFNGLBUFFERDATAARBPROC)eglGetProcAddress("glBufferDataARB");
pMe->glBufferSubData =
(PFNGLBUFFERSUBDATAARBPROC)eglGetProcAddress("glBufferSubDataARB")
if (pMe->glGenBuffersARB == NULL ||
    pMe->glDeleteBuffersARB == NULL ||
    pMe->glBindBufferARB == NULL ||
    pMe->glBufferDataARB == NULL ||
    pMe->glBufferSubDataARB == NULL)
{
   return FALSE;
}
```

8.1.2 Initializing a VBO

Sample Code:

```
GLuint nVBOBufSize = nVertSize + nNormSize;
/*
* calculate VBO offsets. These are needed when setting up the data pointers
*/
pMe->nVertexOffset = 0;
pMe->nNormalOffset = pMe->nVertexOffset + nVertexSize;
/*
* generate a single VBO buffer
*/
pMe->pVBOArrayId = (GLuint*)MALLOC(sizeof(GLuint) * 1);
pMe->glGenBuffersARB(1, pMe->pVBOArrayId);
pMe->glBindBufferARB(GL_ARRAY_BUFFER_ARB, pMe->pVBOArrayId[0]);
/*
* create the empty VBO buffer
*/
pMe->glBufferDataARB(GL_ARRAY_BUFFER_ARB, nVBOBufSize, NULL,
GL_STATIC_DRAW_ARB);
/*
* add the data to the VBO buffer
*/
pMe->glBufferSubDataARB(GL_ARRAY_BUFFER_ARB, pMe->nVertOffset, nVertSize, pMe-
>pVerts);
pMe->glBufferSubDataARB(GL_ARRAY_BUFFER_ARB, pMe->nNormOffset, nNormSize, pMe-
>pNorms);
/*
* unbind VBO buffer untill we are ready to use it
*/
pMe->glBindBufferARB(GL_ARRAY_BUFFER_ARB, 0);
```

8.1.3 Drawing with a VBO

Sample Code:

```
/*
* bind the right VBO buffer
*/
pMe->glBindBufferARB(GL_ARRAY_BUFFER_ARB, pMe->pVBOArrayId[0]);
```

```
/*
* setup the vertex data pointers. Instead of using vertex data pointers we
* use offsets into the vertex data buffer.
*/
glVertexPointer(3, GL_FIXED, 0, (void*)pMe->nVertexOffset);
glNormalPointer(GL_FIXED, 0, (void*)pMe->nNormalOffset);
/*
* unbind the VBO buffer (VBO use is enabled untill bindbuffers is called with
Id=0)
*/
pMe->glBindBufferARB(GL_ARRAY_BUFFER_ARB, 0);
/*
* Make the drawcall as your would using regular vertex data
*/
glDrawElements(GL_TRIANGLE_STRIP, nNumIndices, GL_UNSIGNED_SHORT, pMe-
>pIndices);
```

8.2 Texture Compression

The Zeebo supports ATITC texture compression format. The ATITC texture compression provides huge benefits by allowing more textures to fit in the allotted texture memory, reducing bandwidth usage and providing better texture cache behavior.

Compression ratios:

- 6 : 1 for GL_RGB to GL_COMPRESSED_RGB_ATI_TC (24 bit -> 4 bit/pixel);
- 4 : 1 for GL_RGBA to GL_COMPRESSED_RGBA_ATI_TC (32 bit -> 8 bit/pixel);

To use a compressed image in OpenGL ES, simply replace your glTexImage2D call with the following:

```
glCompressedTexImage2D(
    GL_TEXTURE_2D,
    <miplevel>,
    <GL_COMPRESSED_RGB_ATI_TC | GL_COMPRESSED_RGBA_ATI_TC>,
    <uncompressed image width>,
    <uncompressed image height>,
    0,
    <compressed image size>,
    <compressed image pixels>);
```

To create an ATITC compressed texture you can use one of the following tools.

8.2.1 QXTextureConverter

The QXTextureConverter Library provides a variety of image conversion procedures that allows easy conversion between standard image file formats like Windows Bitmaps (BMP) and Targa (TGA) files into optimized OpenGL ES or D3D Mobile texture formats. The library handles the conversion of the data layout of the input images, whether swizzled or inverted, as well as outputting these images into many texture file formats including ATITC

compressed textures and palettized textures. The library also contains some specialized texture conversion routines like mipmap scaling and normal map generation.

QXTextureConverter is part of QXEngine. See section 2.4.1 for more information about QXEngine.

8.2.2 The Compressonator

The Compressonator (http://www.ati.com/developer/compressonator.html) is a stand alone application that can load textures, compress them to ATI_TC and save them to file with the extension ".atitc". These files start with the following 20 bytes header:

```
typedef struct _ATITC_HEADER
{
   unsigned int signature;
   unsigned int width;
   unsigned int height;
   unsigned int flags;
   unsigned int dataOffset;
} ATITC_HEADER;
```

This header is not part of the ATITC texture and should be stripped before using the texture in OpenGL ES.

The Compressonator can also be used from the command line. For more information type "TheCompressonator –help" on the command line.

8.2.3 ATI Compress

ATI_Compress (http://www.ati.com/developer/compress.html) is ATI's library for texture compression. It is used by The Compressonator. It is also available to third party developers who wish to incorporate it within their own content creation pipeline.

8.3 Rescaling OpenGL ES Rendering Surfaces

OpenGL ES rendering surfaces can be rescaled, allowing significant gains in performance and making easier the porting from existing games written in BREW to Zeebo. The rescaling is done using QUALCOMM_surface_scale.

This OpenGL ES extension enables rescaling an EGL window surface as the surface contents are copied to the target display device while posting the buffer. Both upscaling and downscaling are supported. This extension can only be applied to non-current surfaces.

The initial dimensions of the EGL surface will match the target display, which is VGA (640x480) for Zeebo. The user specifies one or two rectangular regions known as the source and destination rects. The source rect is scaled as required to the size specified in

the destination rect. Usually, the source rect will be less than or equal to the size of the EGL surface.

A larger source rect, up to the implementation limit, may be specified when downscaling is desired. Downscaling can be used to achieve a “fake” anti-aliasing for rendering engines which have no internal anti-aliasing capability. For reasonable results, this will require the rendering engine to render an image which is at least twice the desired size in both dimensions.

The source rect specifies the rectangular area of an EGL window surface that will be rescaled to the destination rect. The source rect may be passed as NULL if the full surface is to be used as the input. If the source rect is larger than the original EGL surface dimensions, the surface is resized which results in a larger rasterization load on the 3D pipeline. To resize the surface, the associated buffers must be deleted and reallocated to accommodate the larger dimensions; this operation may fail if there is insufficient memory or the specified source rect exceeds the implementation limits. If the source rect is smaller than the original surface dimensions, the surface buffers may be deleted and reallocated by the driver to save memory and reduce binning penalties. To determine the (possibly) new dimensions of the EGL surface use eglQuerySurface.

Using smaller window surface dimensions can save render time or, equivalently, increase sustainable frame rate. On games where a memory constrained rendering engine cannot operate on surfaces of the desired dimension, upscaling smaller window surfaces can produce VGA images on the display output.

The destination rect specifies the dimensions to which the entire (possibly resized) EGL window surface will be scaled as a post processing operation. The destination rect may be passed as NULL if the original full surface size is the desired output. The destination rect must be less than or equal to the size of the Zeebo display output, which is VGA.

Find bellow a sample code for upscaling a QVGA (320x240) surface to the full native VGA window surface.

```
// The objective of this sample is to reduce the rasterization
// load on the 3D pipeline by decreasing the surface area it
// renders.  The smaller surface is upscaled to the full
// VGA display size during the buffer swap.
//
// The initial window surface is VGA size to match the display
//

// Do required EGL initialization here ...

// Create the VGA size window surface
window = eglCreateWindowSurface( dpy, config, VGA_window, NULL );

// Any additional EGL setup ..

// Set the window surface to upscale from a QVGA sized region
// to the full VGA display
EGLSurfaceScaleRect src_rect = {0, 0, 320, 240};
eglSetSurfaceScaleQUALCOMM( dpy, window, &src_rect, NULL );
eglSurfaceScaleEnableQUALCOMM( dpy, window, EGL_TRUE );
```


Zeebo Confidential and Proprietary

```
// Setup OpenGL ES rendering to only the VGA region used
glViewport( 0, 0, 320, 240 );
glScissor( 0, 0, 320, 240 );

// Setup other clients to restrict rendering as necessary...

// Draw calls here

// Swap to the display, the content will be upscaled to VGA
eglSwapBuffers( dpy, window, window, ctx );
```


Zeebo Confidential and Proprietary

9 Zeebo Audio

This chapter describes specific audio capabilities that are available on the Qualcomm MSM7201A chipset. Notes on the architecture, supported audio formats and tradeoffs when using audio are also provided.

The MSM7201A chipset offers a wide range of audio features and more robust performance of 3D graphics with audio. All graphics processing are offloaded to the external Adreno 130 graphics core. This leaves the DSP completely free for audio processing. This design is geared for gaming and advanced audio applications.

Qualcomm CMX provides game and application developers with a wide array of high-quality audio options to greatly enhance the game player experience and add significant value to their games. The supported audio features for use with games or applications are:

- Direct playback and control of audio objects – MIDI-based audio objects (MIDI, XMF, and PMD) and encoded audio objects (PCM, ADPCM, QCP, and MP3);
- Simultaneous playback of multiple MIDI objects and multiple encoded audio objects (including PCM, ADPCM, and QCP);
- Global loading and unloading of DLS;
- QAudioFX, the 3D audio solution, including features such as positional sound, rolloff, reverberation, and Doppler effects;

9.1 Playback of MIDI and encoded audio objects

Qualcomm's CMX solution includes the ability to play back and control an audio object. The file formats that are currently supported include MIDI, PMD, XMF, QCP, ADPCM, PCM, and MP3.

The supported encoded audio and MIDI file formats for single playback are presented in the following subsections.

9.1.1 Supported encoded audio formats

The supported encoded audio file formats are described in this section. Choosing a file format for a particular use is entirely up to the developer; however, it should be noted that there are preferred file formats for certain uses. This information is also included in the following subsections.

9.1.1.1 QCELP (.qcp)

CMX supports 13 kbps fixed full rate QCELP files at 8 kHz. The format, a codec originally intended for voice, is ideally suited for voice audio recordings and sound effects (sounds that do not have high-frequency content). This audio format may not be well suited for melodic music.

Each QCP object consumes approximately 2% of the total DSP load utilization. QCP objects must be encoded as fixed full rate. QCP objects encoded at other rates will not play back correctly.

The CMX Studio authoring tool is capable of exporting compatible QCP files as long as the sampling rate of the original imported audio is 8 kHz.

9.1.1.2 ADPCM (.wav)

Adaptive Differential Pulse Code Modulation (ADPCM) uses a compression technique that records the differences between samples and adjusts the coding scale to accommodate for large and small differences. ADPCM files with 4-bit samples are one-quarter the size of linear PCM files that has 16-bit samples. ADPCM is a recommended compression format for music; it has a larger file size than QCP, but is higher quality sound.

Each ADPCM object consumes approximately 2% of the total DSP load utilization. The MSM7201A supports IMA ADPCM, 4 bits per sample, 4, 8, 12, 16, 20, 24, 28, 32, 36, 44.1, 48 kHz. IMA ADPCM is a common ADPCM encoding format. When playing back multiple ADPCM objects simultaneously, each object can have a different sampling rate.

The CMX Studio authoring tool is capable of exporting compatible IMA ADPCM files as long as the sampling rate of the imported audio file is not changed.

9.1.1.3 PCM (.wav)

Pulse Code Modulation (PCM) files are raw, uncompressed waveform samples. Linear PCM is also more ideally suited to musical audio than is the QCP format. These files can be very large.

Each PCM object consumes approximately 2% of the total DSP load utilization. The MSM7201A and chipset supports linear PCM, mono or stereo, 8 or 16 bits per sample, at 4, 8, 11.025, 12, 16, 22.05, 24, 32, or 44.1 kHz.

9.1.1.4 MP3 (.mp3)

MPEG-1, Audio Layer 3, more commonly known as MP3, is a popular digital audio format that represents PCM-encoded audio utilizing a lossy compression technique. The compression method is based on psychoacoustic models that eliminate components of audio that are not audible to the human ear, thereby reducing memory requirements.

The MSM7201A chipset supports all bitrates and sampling rates. It also has the ability to play MP3 in conjunction with 3D graphics.

9.1.2 Supported MIDI file formats

MIDI playback occur through Qualcomm's CMX synthesizer and wavetable. Playback of a single MIDI object (not simultaneous with other MIDI objects) is through the HQ synthesizer. For PMD files, the selection of the HP or HQ synthesizer, or a combination of the two, is based on the polyphony mode setting within the file. If playback is simultaneous with other MIDI objects, then all files play back through the HP synthesizer. In this case, the polyphony mode setting of PMD files is ignored.

9.1.2.1 MIDI (.mid)

CMX supports SP-MIDI and General MIDI level 1 and 2 Standard MIDI Format (SMF). The SMF file format stores MIDI data and other data typically needed by a sequencer. At a minimum, a MIDI representation of a sound includes values for the note's pitch, length, and volume and may also include additional characteristics, such as attack and delay time. One SMF file can store information for numerous patterns and tracks and was designed to be generic so that the information can be used by any sequencer.

MIDI playback is through Qualcomm's CMX synthesizer and wavetable. This product actually contains two synthesizers. The High Quality (HQ) synthesizer is used for better sounding output but requires more DSP utilization per voice. The High Polyphony (HP) synthesizer is less complex and better suited for playing back multiple MIDI files simultaneously.

MIDI files play back through the HQ synthesizer and wavetable by default when played individually. When played simultaneously with other MIDI objects, they are played through the HP synthesizer.

MIDI can also be played using MIDI messages which are sent directly to the core synthesizer using BREW API calls. These can also be used simultaneously with MIDI files and/or encoded audio objects. The behavior and usage of the BREW APIs for MIDI messages is discussed in greater detail in section 9.1.2.4.

9.1.2.2 XMF (.xmf and .mxmf)

The eXtensible Music Format (XMF) and its mobile version (mXMF) serve as wrappers for MIDI and DLS files. All features and restrictions related to MIDI files also apply to XMF and mXMF files.

XMF and mXMF files, which may contain DLS, play back through the HQ synthesizer and wavetable when played individually.

When playing XMF and mXMF files simultaneously with other MIDI objects, all instruments play back through the HP synthesizer. Also, DLS instruments within an XMF file, if any, play

back as General MIDI instruments. See section 9.1.2.4 for more details on using DLS with XMF files.

9.1.2.3 CMX (.pmd)

The CMX file format is a wrapper format that can contain a MIDI file, an encoded audio file, LED control, vibration, images, animation, text, text wipe, or any combination of these features. One of the most significant advantages of the CMX file format is its time-synchronization capability. This capability provides developers with the ability to synchronize the included contents together. This can also be used to synchronize vibration and LED control with audio playback. Another application of this feature is for time-synchronized cut scenes that include synchronized text, still images (JPEG or PNG), and animation.

CMX files, hereafter referred to by their extension PMD, can be created using Qualcomm's CMX Studio authoring tool or the CMX Batch Converter Tool.

During the creation of a PMD file, imported MIDI objects have all instruments set to the HQ bank by default. The CMX Studio authoring tool enables switching particular instruments to the HP bank to play back through the HP synthesizer.

DLS instruments also play back through the HQ synthesizer. Using HQ and DLS reduces the total number of voices available because the synthesizer is more complex.

When playing PMD files simultaneously with other MIDI objects, all instruments play back through the HP synthesizer, regardless of the instruments' bank setting. In this case, DLS instruments within a PMD file play back as General MIDI instruments. See section 9.1.2.4 for more details on using DLS with PMD files.

The MSM7201A chipset supports the following objects within a PMD file:

One MIDI object (with DLS) + one ADPCM or QCP object + LED + vibrator

NOTE: If two or more PMD files with LED/vibration are played at the same time, the LED/vibration command call in the second file will be ignored. The Zeebo wireless gaming platform does not support LED control and vibration.

9.1.2.4 DLS (.dls)

The MSM7201A chipset support Downloadable Sounds (DLS) as part of PMD and XMF files. The embedded DLS will play back through the HQ synthesizer when the PMD or XMF is played back individually. However, when the PMD or XMF file is played simultaneously with other MIDI objects, DLS instruments will play back as General MIDI instruments.

The MSM7201A chipset also support standalone DLS files when used in conjunction with MIDI file playback. The global DLS load/unload BREW API can be used to load a DLS. Any MIDI object can play the DLS instrument by making the proper bank MSB, LSB, and program changes as specified by the DLS file.

Because the same DLS can be made available to multiple MIDI objects, using DLS reduces the memory requirements of an application. Very short and simple sound effects can be achieved using DLS.

9.2 Multisequencing – Simultaneous playback of audio objects

Qualcomm's CMX technology supports the playback of multiple, simultaneous MIDI and certain encoded audio objects. The term *multisequencer* is used throughout this document to refer to the simultaneous playback of multiple audio objects. It can be used to combine the playback of specific audio objects based on game events and user input.

This section delineates the details of the multisequencer and the limitations on this feature. The limitations on the number and audio format of the objects depend on the chipset being used within the target device. Multisequencing is achieved using BREW APIs, which are described in section 9.5.

The MSM7201A chipset is capable of simultaneous playback of the following:

- Four MIDI objects + four encoded audio objects;
- The encoded audio objects can be QCP, ADPCM, or PCM with QAudioFX 1.1;
- The encoded audio objects can be QCP, ADPCM, PCM, or MP3 with QAudioFX 1.1c;
- The four encoded object formats can be mixed and matched;

A maximum of four MIDI objects can be played simultaneously independent of the number of encoded audio objects. Likewise, up to four encoded audio objects can be played simultaneously regardless of the number of MIDI objects playing.

The sections below details the limitations and restrictions when using simultaneous audio on the MSM7201A chipset.

9.2.1 Restrictions with MP3 and simultaneous audio

The MSM7201A contains four audio decoders that are allocated in order to play encoded audio objects as illustrated in the figure below.

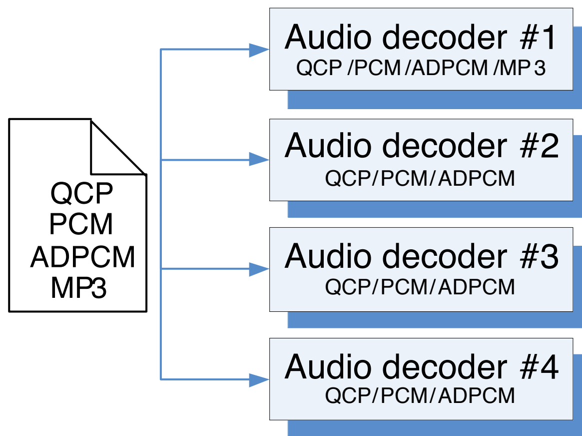


Figure 9-1 Audio decoders on MSM7201A

MP3 decoding is supported only with audio decoder #1. Depending on the sequence of events in an application, audio decoder #1 could be assigned to decode QCP, PCM, or ADPCM instead. If this happens and playback of MP3 is attempted, an error is returned and the results may be unpredictable. Similarly, attempting playback of a second MP3 when an MP3 is already playing will result in an error and the results may be unpredictable.

An audio resource manager must be implemented in applications that mix MP3 with QCP, PCM, or ADPCM. Refer to Section 9.2.3 for details on the MSM7201A's audio decoder allocation algorithm.

9.2.2 Exceeding number of resources with simultaneous audio

Developers should ensure that their applications do not attempt to play a fifth MIDI file when four MIDI files are already playing. Similarly, playing a fifth QCP, ADPCM, or PCM file when four of them are already playing should not be attempted. The results may be unpredictable.

If there is a possibility that the application could play more than four simultaneous files, implementation of an audio resource manager is required. This allows the application to ensure that the resources within the driver are not exceeded. The resource manager can keep track of the number of files which are currently playing at halt one to accomodate a new playback when necessary.

9.2.3 Decoder allocation algorithm with simultaneous audio

An audio resource manager must be implemented in applications that mix MP3 with QCP, PCM, or ADPCM. It should also be implemented if there is a possibility that the application could play more than four MIDI files or four QCP, PCM, or ADPCM files, even without MP3. The resource manager must abide by the limitations and restrictions of the MSM7201A audio driver so that each new playback attempt is guaranteed to succeed.

When playback is initiated, the MSM7201A driver will always use the lowest-numbered unassigned decoder. When playback ends, the decoder is freed and made available for a new playback. As mentioned in Section 5.1.4.1, MP3 can only play on audio decoder #1.

The following figures illustrate the MSM7201A audio driver's decoder allocation algorithm. M1 and M2 are MP3 files. A1 through A8 are ADPCM .wav files. In all of these cases, QCP and PCM files can be substituted for ADPCM files. The allocation of the MIDI synthesizers is identical to that of the audio decoders and is therefore not illustrated here.

In all cases below, each file is either stopped or allowed to play to completion. The result is the same regardless of the end-of-playback condition.

Figure 9-2 illustrates the audio decoder allocation when starting ADPCM files at various points in time in response to real-time game events. The application does not attempt to play more than four ADPCM files simultaneously.

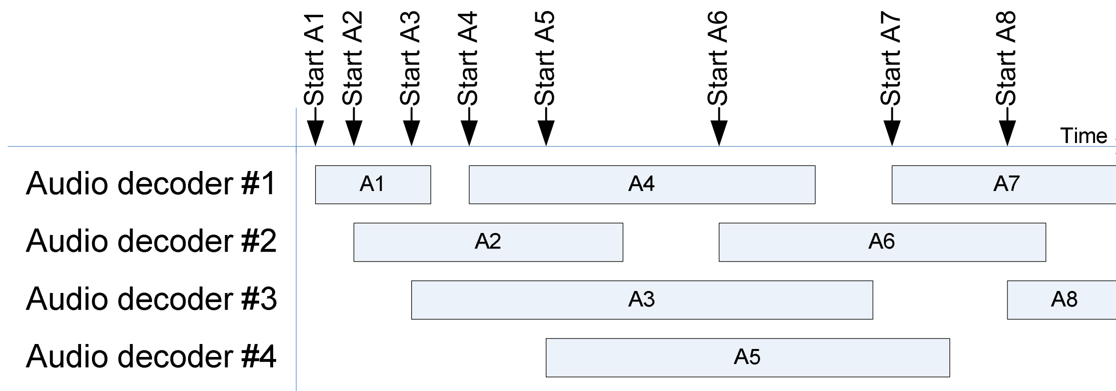


Figure 9-2 Decoder allocation with multiple ADPCM

Figure 9-3 illustrates the audio decoder allocation that results when an application attempts to keep audio decoder #1 always allocated to an MP3 background track. Sound effects are played on audio decoders #2, #3, and #4. When the MP3 file ends, the application always prioritizes the start of another MP3 file before the next ADPCM play.

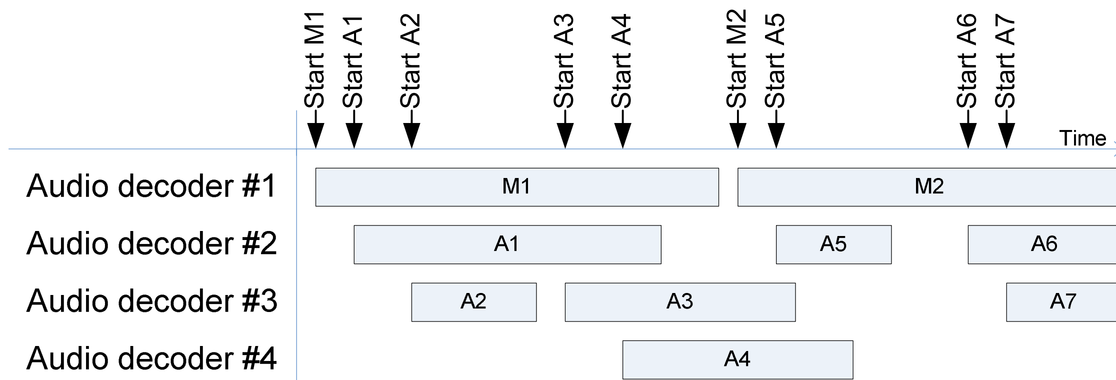


Figure 9-3 Background MP3 music with foreground ADPCM sound effects

Zeebo Confidential and Proprietary

Figure 9-4 illustrates the audio decoder allocation that results when an application plays a combination of MP3 and ADPCM files. The application is careful to ensure that audio decoder #1 is always free before attempting any MP3 playback.

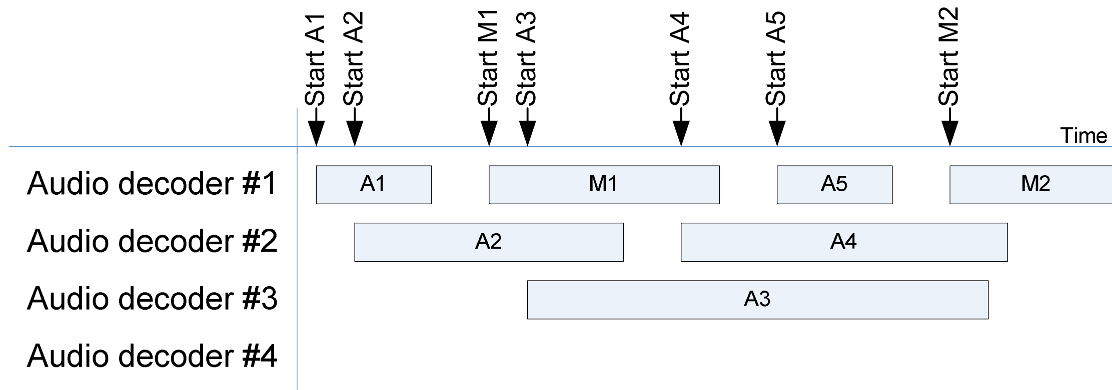


Figure 9-4 Decoder allocation when mixing MP3 and ADPCM

The following cases illustrate playback attempts which violate the limitations and restrictions of the MSM7201A audio driver.

In Figure 9-5, the application attempts to play a fifth ADPCM file when four are already playing. No free decoder is available of the fifth ADPCM file. This is prohibited. The application should wait until one of the ADPCM files has completed before starting the fifth. Alternatively, the application can choose to stop one of the existing ADPCM file in favor of starting the fifth.

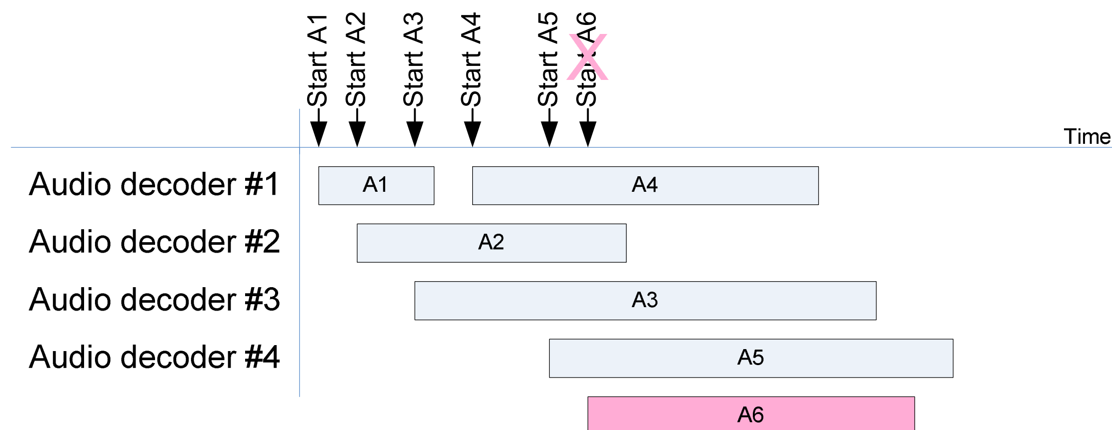


Figure 9-5 Playing too many ADPCM simultaneously (prohibited)

In Figure 9-6, the application attempts to play a fourth ADPCM file when one MP3 file and three ADPCM files are already playing. No free decoder is available for the fourth ADPCM file. This is prohibited. Similar to the previous scenario, the application should wait until one of the audio files has completed or stop one of them to start the fourth ADPCM file.

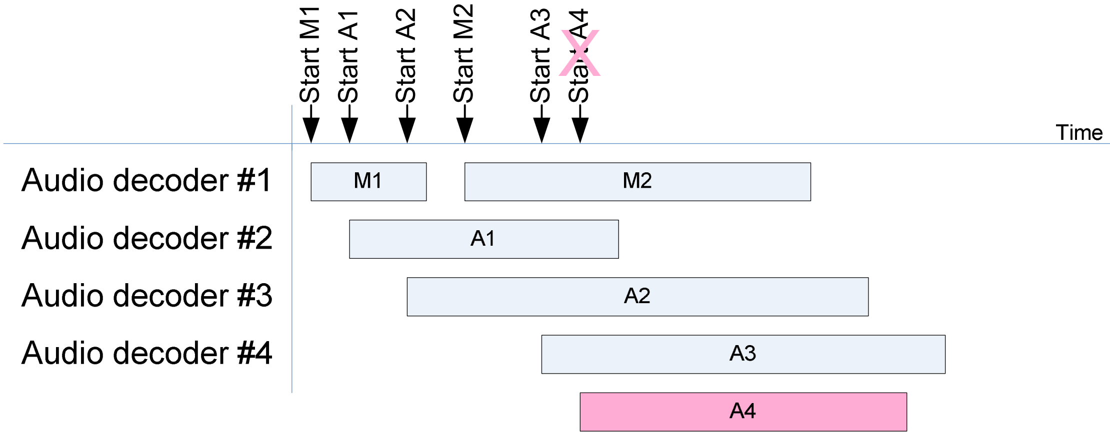


Figure 9-6 Playing too many ADPCM with MP3 simultaneously (prohibited)

In Figure 9-7, the application attempts to play an MP3 file when audio decoder #1 is not available. This is prohibited. The application should wait until audio decoder #1 is available or stop the ADPCM which is playing on it before starting the MP3 file.

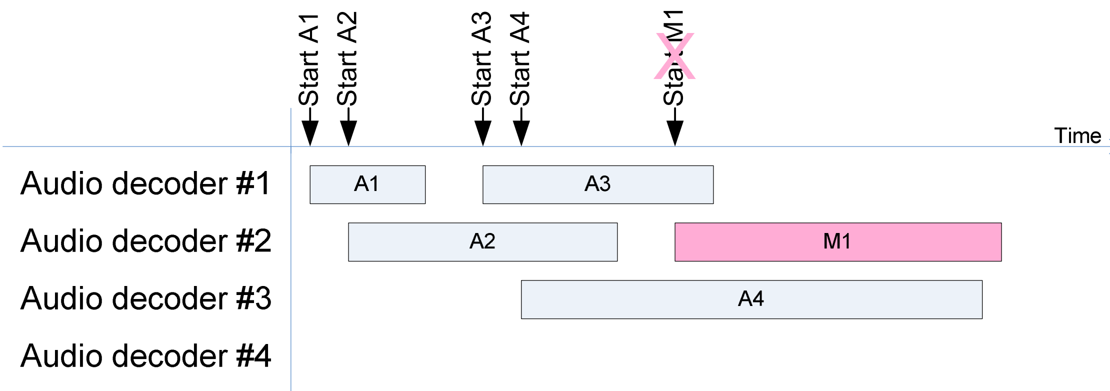


Figure 9-7 Playing MP3 when audio decoder #1 is not available (prohibited)

In Figure 9-8, the application attempts to play a second MP3 file when one is already playing. In this case, audio decoder #1 is also not available. This is prohibited. Similar to the previous scenario, the application should wait until audio decoder #1 is available or stop the MP3 which is playing on it before starting the second MP3 file.

Zeebo Confidential and Proprietary

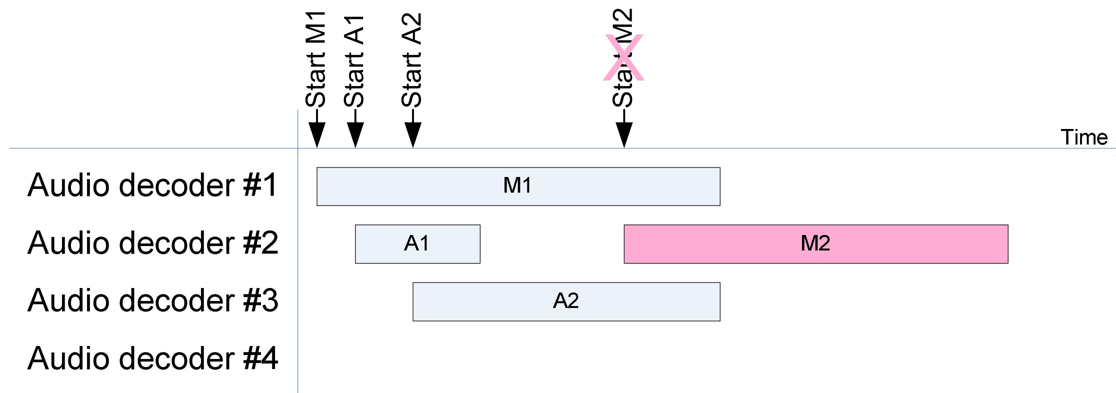


Figure 9-8 Playing two MP3 files simultaneously (prohibited)

9.3 QAudioFX – 3D audio

QAudioFX is Qualcomm's positional audio solution and is available on MSM7201A chipset. This option enables developers to position and move sounds within a 3D environment. Movements and sound point origination can be determined by dynamic game events, including game character movements, as well as by game player input.

QAudioFX is not available for this first release of Zeebo wireless gaming platform but it is in the roadmap for 2009.

In the current implementation, up to four ADPCM, PCM, or QCP media objects can be positioned in a 3D environment. The playback of these four media objects can also be used in conjunction with the multisequencer capability to play up to four MIDI objects. Positioning MIDI is not currently supported. In addition, roll off can be used to create realistic effects as objects are repositioned closer or father away from the listener.

Additional effects that are included in QAudioFX 1.1, such as Doppler and Reverb, can also be achieved by using the listener, environment, and sound source settings. QAudioFX on MSM7201A and equivalent chipsets use version 1.1c to indicate the capability of including MP3 file in multisequencing (section 9.2). Details about the BREW API call sequence and usage for QAudioFX are included in section 9.5.

There are specific requirements that must be met by an implementation in order to achieve the desired 3D audio effects which are described in the following sub sections.

9.3.1 Audio objects and environments

Audio objects and an environment comprise the 3D world that must be created in order to achieve 3D audio. Audio objects are single entities that have an associated media format and source. Multiple audio objects can then be created and attached to an environment that is the 3D world around the listener. Objects that are added to the environment are affected by the 3D settings of the environment. Objects that remain detached from the environment should play back without any 3D audio effects.

In addition, there are enable/disable flags for the audio objects and the environment. The availability of these flags varies on a feature-by-feature basis. By enabling features and setting parameters for the objects and the environment, the application can efficiently apply several 3D characteristics to all audio objects that are attached to the environment.

NOTE: Currently, only one environment can be created at a time.

The required state of the object and environment for all available QAudioFX features is described in the following section.

9.3.2 Positional audio and rolloff effects

Positional audio and rolloff effects are used when it is desired to give audio playback direction. Sounds that are positioned are perceived as though they were originating from a particular location. Rolloff is used to attenuate a sound source according to its distance from the listener.

To get positional audio and rolloff effects, the application must add all objects for which this feature is desired to the environment. Furthermore, the 3D flag (the enable flag for positional, rolloff and Doppler) must be enabled explicitly for the audio objects and for the environment.

NOTE: There is a different enable flag for reverb effects.

The application can set the listener and object positions. Both default to the origin upon initialization. Default values are used for rolloff settings. The specific default values will be described in section 9.5. Positional and rolloff settings can be updated at any time, regardless of whether 3D is enabled, whether the audio object has been added to the environment, or the state of playback. However, the effects during playback would only be realized if all necessary requirements are met.

9.3.3 Reverberation effects

Reverberation effects are used when it is desired for the audio playback to sound as though it was occurring in a particular environment. To achieve this effect, the audio objects for which reverb effects are desired must be attached to the audio environment. Furthermore, the reverb level for each of these objects must be explicitly set to a nonminimal value, and reverb for the environment must be enabled. There is no flag to enable/disable reverb on a per object basis. Instead, the reverb gain is configured to achieve the same result.

There are several presets that can be used in creating reverberation effects. The reverb gains, decay times, and damping factors can then be further altered by the application. The reverb presets that are currently available include the following: Auditorium; Room; Bathroom; Concert hall; Cave; Arena; Forest; City; Mountains; Underwater; Alley; Hallway; Hangar; Living room; Small room; Medium room; Large room; Medium hall; Large hall; Plate

9.3.4 Doppler effects

Doppler effects can be used when you want to shift the frequency of audio playback for moving objects. This is done by creating a relative velocity between a sound object and the listener. In order to enable this effect, the same rules as those for achieving positional audio apply. In fact, enabling 3D on the environment and the attached objects that have a relative velocity with respect to the listener will exhibit Doppler shifts during playback.

9.4 Summary of audio support on Zeebo

The audio capabilities are not limited by concurrency with 3D graphics and include:

- Simultaneous playback of four MIDI or PMD objects (totaling up to 72 polyphony voices) + four ADPCM, QCP or PCM objects;
- PMD (each PMD can include one MIDI object with DLS + one ADPCM or QCP object + LED + vibrator):
    - Individual components within the PMD file are counted in the total number of encoded audio objects that can be played back simultaneously.
    - One PMD file uses one MIDI object resource, regardless of whether MIDI is embedded in the PMD or not. For example, it is not possible to play back four PMD with four additional MIDI, even if the PMD objects do not contain MIDI.
    - PMD with DLS cannot be used if other MIDI objects are to be played back simultaneously. If attempted, General MIDI instruments will play back instead of DLS instruments.
- Single-file audio formats:
    - 72-polyphony MIDI (total);
    - ADPCM - IMA ADPCM; 8, 16, or 32 kHz; 4 bits per sample
    - QCP – Fixed full rate only;
    - Linear PCM – Mono or stereo;
    - MP3;
- Global DLS loading for use with MIDI-based playback;
- MIDI messaging for direct access to the MIDI synthesizer within CMX;
- QAudioFX can be used for achieving 3D audio effects on ADPCM, PCM, QCP, or MP3 files that include:
    - Positional audio;
    - Rolloff;
    - Reverberation effects;
    - Doppler effects;

Picture summarizes the audio support on Zeebo.

Format	Encoding	Sampling rate	Single playback	Multiple playbacks	Positional	Reverb	Doppler
PCM	8-bit	4.000 to 44.100 kHz (in single Hz increments) and 48 kHz	Yes (mono or stereo)	Yes (mono or stereo)	Yes (mono only)	Yes (mono or stereo)	Yes (mono only)
	16-bit	4.000 to 44.100 kHz, and 48 kHz	Yes (mono or stereo)	Yes (mono or stereo)	Yes (mono only)	Yes (mono or stereo)	Yes (mono only)
ADPCM	4-bit	4.000 to 44.100 kHz, and 48 kHz	Yes (mono or stereo)	Yes (mono or stereo)	Yes (mono only)	Yes (mono or stereo)	Yes (mono only)
QCP	Any	8 kHz	Yes (mono only)	Yes (mono only)	Yes (mono only)	Yes (mono only)	Yes (mono only)
MP3	Any	Any	Yes	Yes (QAudioFX 1.1c and later)	Yes (QAudioFX 1.1c and later)	Yes (QAudioFX 1.1c and later)	Yes (QAudioFX 1.1c and later)
MIDI	—	—	Yes	Yes	No	No	No
PMD (MIDI + QCP/ ADPCM)	—	—	Yes	Yes	No	No	No
XMF (MIDI + DLS)	—	—	Yes	Yes	No	No	No

Figure 9-9 Audio support on Zeebo

9.5 BREW APIs for Audio

All available BREW IMedia and IDLS APIs are defined in the BREW_API_REFERENCE document that is included in the installation of the BREW SDK. This section highlights the BREW APIs that are required to perform simultaneous playback of multiple media objects, implement DLS loading and unloading, send MIDI messages, and achieve 3D audio using QAudioFX BREW APIs.

9.5.1 Playing multiple audio objects simultaneously

The MSM7201A chipset supports up to four MIDI + 4 QCP/ADPCM/PCM/MP3. The BREW Simulator supports up to one MIDI + 4 QCP/ADPCM/PCM.

9.5.1.1 BREW API call sequence for simultaneous playback

Sample code is included at the end of this section to elucidate the sequence of events for using Qualcomm’s channel share (multisequencer) API. The following series of API calls can

be used to simultaneously play back multiple MIDI and ADPCM files. Other encoded audio file types are supported by simply changing the class ID in the sequence below.

1. Create each MIDI media object using one of the following two API call sequences:
   a. Call ISHELL_CreateInstance(AEECLSID_MEDIAMIDI). Load each MIDI file using IMEDIA_SetMediaData().
   b. Call IMEDIAUTIL_CreateMedia().
2. Share each MIDI media object via IMEDIA_EnableChannelShare() if there are multiple MIDI objects. This API call is extremely important and must be called on all ADPCM objects if concurrent playback of MIDI or other ADPCM objects is desired.IMEDIA_EnableChannelShare() calls separately for each object and is optional if the object called is not going to be played back concurrently with other MIDI objects. In other words, if there are multiple MIDI objects, IMEDIA_EnableChannelShare() must be called for each object with the following exception: if there is one MIDI object plus four ADPCM objects, IMEDIA_EnableChannelShare() must be called on the ADPCM objects, but not on the MIDI objects.
3. Create audio objects for each ADPCM segment using one of the following two methods:
   a. Call ISHELL_CreateInstance() with class ID AEECLSID_MEDIAADPCM as appropriate. Load each encoded media file via IMEDIA_SetMediaData().
   b. Call IMEDIAUTIL_CreateMedia().
4. Share each encoded audio object via IMEDIA_EnableChannelShare(). IMEDIA_EnableChannelShare() should be called immediately after the objects are created.
5. Play each object via IMEDIA_Play(). In a new session, on the first call to IMEDIA_Play(), the application must wait for MM_STATUS_START to be returned before calling IMEDIA_Play() on another object. A new session starts whenever no media objects have been created or whenever all media objects have been freed via IMEDIA_Release(). If concurrent playback of QCP with MIDI is desired, the first call of a new session to IMEDIA_Play() must be a QCP object. In other words, call IMEDIA_Play() on QCP, then IMEDIA_Play() on MIDI. This restriction does not exist for ADPCM, PCM or MP3. To switch between playbacks of multiple QCP to multiple ADPCM or vice versa, a new session must be started. All media objects must be freed via IMEDIA_Release() before creating the new media objects.

The following code snippet contains two functions: LoadOneMedia() and MediaNotify(). The first function is used to load and play a media file, while the second calls the first to load and play multiple media files.

Example:

```
// **********************************************
// **********************************************
```


Zeebo Confidential and Proprietary

```
// Multi-Sequencing Snippet
// ********************************************
// ********************************************

// Note: Error handling removed for brevity
// Routine to load and play a media file
boolean LoadOneMedia(char *pFileName, IMedia **ppIMedia,
                     boolean fChannelShare)
{
   int iReturn;

   if(pFileName == NULL || ppIMedia == NULL)
      return false;

   // First figure out what type of media to create
   // (This is an internal routine and source code is
   // not part of this example.)
   AEECLSID WhichCLSID = AEECLSID_MEDIAMIDI;
   WhichMediaType(pFileName, WhichCLSID);

   iReturn = ISHELL_CreateInstance(m_pIShell, WhichCLSID,
                                (void **)ppIMedia);

   AEEMediaData   MediaData;
   MediaData.clsData = MMD_FILE_NAME;
   MediaData.pData = pFileName;
   MediaData.dwSize = 0;

   iReturn = IMEDIA_SetMediaData(*ppIMedia, &MediaData);

   if(fChannelShare)
      {
      iReturn = IMEDIA_EnableChannelShare(*ppIMedia, true);
      }
   // Need a pointer to IMedia object for the RegisterNotify parameter so
   // media can be stopped and restarted.
   iReturn = IMEDIA_RegisterNotify(*ppIMedia, MediaNotify, *ppIMedia);

   // Make sure the file is playing
   int iMediaState;
   boolean fIsChanging;
   iMediaState = IMEDIA_GetState(*ppIMedia, &fIsChanging);
   if(iMediaState != MM_STATE_PLAY)
      {
      iReturn = IMEDIA_Play(*ppIMedia);
      }

   return true;
}


// For example, start with 4 QCP files.
// Start the first one playing.
m_fFirstStatusStart = false;
fReturn = LoadOneMedia(QCP_FILE_1, &m_pIMediaSounds[0]);

// Must now wait until receiving MM_STATUS_START on
// the first object before starting other objects.
```

```
void MediaNotify(void *pUser, AEEMediaCmdNotify *pCmdNotify)
{
   int      iReturn;
   IMedia   *pIMedia = (IMedia *)pUser;
   // Only care about the play commands
   if(pCmdNotify->nCmd == MM_CMD_PLAY)
      {
      switch(pCmdNotify->nStatus)
         {
         case MM_STATUS_START:
            if(!m_fFirstStatusStart && pIMedia == m_pIMediaSounds[0])
               {
               m_fFirstStatusStart = true;
               if(fReturn)
                  fReturn = LoadOneMedia(QCP_FILE_2, &m_pIMediaSounds[1]);
               if(fReturn)
                  fReturn = LoadOneMedia(QCP_FILE_3, &m_pIMediaSounds[2]);
               if(fReturn)
                  fReturn = LoadOneMedia(QCP_FILE_4, &m_pIMediaSounds[3]);
               }
            break;
         case MM_STATUS_DONE:
            // Start the media back up because we want it going in an
            // infinite loop
            if(pIMedia != NULL)
               {
               iReturn = IMEDIA_Play(pIMedia);
               }
            break;
         }
      }
}
```

The multisequencer operations begin and the playback of these four files starts simultaneously.

9.5.1.2 Playback Control

In order to control the playback (pausing, resuming, seeking, stopping, etc.) for a specific media, invoke the corresponding IMedia API on that IMedia object. To perform the operation on all the media files at once, call the corresponding IMedia API for all the objects.

NOTE: Stopping a media object will stop just that media object playback; other media objects continue to play as part of the multiple sequences.

Stop objects

To stop the multisequence completely, call stop on each object individually:

```
IMEDIA_Stop(pIMediaFoo1);
IMEDIA_Stop(pIMediaFoo2);
IMEDIA_Stop(pIMediaFoo3);
```

Release objects

When the multisequence has completed, release each IMedia object individually:

```
IMEDIA_Release(pIMediaFoo1);
IMEDIA_Release(pIMediaFoo2);
IMEDIA_Release(pIMediaFoo3);
```

9.5.2 Global Loading and Unloading of DLS

Downloadable sounds are custom wavetable instruments for a MIDI synthesizer. Implementing sound effects as a DLS instead of an encoded QCP/ADPCM segment allows for:

- Sounding on and off via MIDI commands;
- Adjustable playback pitch and volume depending on the note on number and velocity;
- Ability to operate on the sound using any supported MIDI command such as the pitch wheel;

Play commands use the globally-loaded DLS if the content performs the proper bank and program changes. For example, if the DLS instrument is located at MSB 20, LSB 0, and Program 13, the MIDI must also perform these bank and program changes in order to play the DLS instrument. This reduces the application size by avoiding having to duplicate the same DLS across multiple PMD or XMF files.

The BREW IDLS APIs allow for global loading and unloading of DLS. This feature is available on the MSM7201A chipset. Currently, DLS cannot be used in conjunction with the multisequencing. Also, note that the DLS feature is not supported in the BREW emulator.

The following two sections discuss the usage of the BREW APIs to perform global loading and unloading of DLS.

9.5.2.1 Global Loading of DLS

In order to globally load and play a DLS media object, the following sequence of APIs must be used:

1. Load the DLS using IDLS_Load();
2. Create the MIDI media object using one of the following two API call sequences:
    a. Call ISHELL_CreateInstance(AEECLSID_MEDIAMIDI). Load each MIDI file using IMEDIA_SetMediaData();
    b. Call IMEDIAUTIL_CreateMedia().;
3. Play the MIDI object via IMEDIA_Play();

Example:

```
// *********************************************
```

```
// *********************************************
// Global Loading DLS Snippet
// *********************************************
// *********************************************

// Note: Error handling removed for brevity

// Do nothing callback routine
static void LoadDLSCB(void *pUsr)
{
   CMyApp *pMe = (CMyApp *)pUsr;
   CALLBACK_Cancel(&pMe->m_cbDLSLoad);
}

// Create the DLS object
int iReturn = ISHELL_CreateInstance(m_pIShell, AEECLSID_MEDIADLS,
                                    (void **)&m_pIDLS);

AEEMediaData      MediaData;
MediaData.clsData = MMD_FILE_NAME;
MediaData.pData = DLS_FILE;
MediaData.dwSize = 0;

CALLBACK_Init(&m_cbDLSLoad, LoadDLSCB, (void*)this);
iReturn = IDLS_Load(m_pIDLS, &MediaData, &m_cbDLSLoad, &m_LoadDLSRet);

// Check if the DLS is globally loaded.
boolean fIsGlobal = false;
iReturn = IDLS_IsGlobal(m_pIDLS, &fIsGlobal);

if(fIsGlobal)
   {
   // Globally loaded.  Play a MIDI file that used the DLS
   LoadOneMedia(MIDI_DLS_FILE, &m_pIMediaMIDIWithDLS, false);
   return;
   }
```

9.5.2.2 Global Unloading of DLS

In order to globally unload and play an existing MIDI media object, the following sequence of APIs must be used:

1. Unload the DLS using IDLS_Unload(). It is important to set the callback function pointer parameter to the function. Not setting this parameter could result in improper DLS unloading.
2. Play the MIDI object via IMEDIA_Play().

Example:

```
// *********************************************
// *********************************************
// Global Unloading DLS Snippet
// *********************************************
// *********************************************
```


Zeebo Confidential and Proprietary

```
// Note: Error handling removed for brevity

// Do nothing callback routine
static void UnLoadDLSCB(void *pUsr)
{
   CMyApp *pMe = (CMyApp *)pUsr;
   CALLBACK_Cancel(&pMe->m_cbDLSUnLoad);
   if (pMe->m_UnLoadDLSRet == SUCCESS)
      {
      }
}

// Make sure the MIDI is stopped before unloading DLS
IMEDIA_Stop(m_pIMediaMIDIWithDLS);

// Check if the DLS is globally loaded.
boolean fIsGlobal = false;
int iReturn = IDLS_IsGlobal(m_pIDLS, &fIsGlobal);
if(fIsGlobal)
   {
   CALLBACK_Init(&m_cbDLSUnLoad, UnLoadDLSCB, (void*)this);
   IDLS_Unload(m_pIDLS, &m_cbDLSUnLoad, &m_UnLoadDLSRet);
   IDLS_Release(m_pIDLS);
   m_pIDLS = NULL;
   }

// Start the play back up with the DLS unloaded
iReturn = IMEDIA_Play(m_pIMediaMIDIWithDLS);
```

9.5.3 Sending MIDI Messages

In addition to playing back MIDI files, the CMX audio engine can handle MIDI messages sent via BREW API calls to dynamically create and change MIDI output sound.

9.5.3.1 Using MIDI Messages Buffers

A maximum of 32 MIDI commands may be issued in one MIDI message buffer. The supported MIDI commands include, but are not limited to, the following:

- Note on (0x90);
- Note off (0x80);
- Control change (0xB0);
- Program change (0xC0);
- Channel after touch (0xD0);
- Pitch wheel (0xE0);

Construct the MIDI command as it would appear in a MIDI file. For a “note on” command for note `0x45` on channel 5 with velocity `0x7f`, the command would be:

```
0x95 0x45 0x7f
```

For more than one command, string them together:

```
0x95 0x45 0x7f 0x95 0x35 0x7f
```

This example issues two notes on commands on channel 5. Note that there is no timing information in the string of commands because all commands are understood to be issued immediately. To send a few note on and off messages streamed in real time, use IMEDIAMIDIOUTMSG_SetMIDIMsg() to set the buffer containing the MIDI command. Also, if an unsupported MIDI command is encountered, the command will fail and no MIDI commands will be issued.

Example:

The following illustrates the use of MIDI commands to create MIDI output sound:

```
// First the MIDI out message must be created using
// AEECLSID_MEDIAMIDIOUTMSG.  If AEECLSID_MEDIAMIDI is
// used, the call to IMEDIAMIDIOUTMSG_SetMIDIMsg will return
// EUNSUPPORTED.
iReturn = ISHELL_CreateInstance(m_pIShell, AEECLSID_MEDIAMIDIOUTMSG,
                                (void **)ppIMedia);

// You MUST call IMEDIA_SetMediaData or you cannot send
// MIDI messages
AEEMediaData   MediaData;
m_MIDIMsgBuf[0] = 0xB0;
m_MIDIMsgBuf[1] = 0x07;
m_MIDIMsgBuf[2] = 0x7F;
MediaData.clsData = MMD_BUFFER;
MediaData.pData = m_MIDIMsgBuf;
MediaData.dwSize = 3;

iReturn = IMEDIA_SetMediaData(*ppIMedia, &MediaData);

iReturn = IMEDIA_RegisterNotify(*ppIMedia, CMIDIMsg::MediaNotify, this);

// The first MIDI message is ignored in the current
// implementation. To get around this,
// call play immediately after calling SetMediaData
iReturn = IMEDIA_Play(*ppIMedia);

// Now, when sending a MIDI command (like turning on
// a note) Do the following.
m_MIDINoteOnMsgBuf[0] = 0x90;
m_MIDINoteOnMsgBuf[1] = 0x3C;
m_MIDINoteOnMsgBuf[2] = 0x7F;
iReturn = IMEDIAMIDIOUTMSG_SetMIDIMsg(m_pIMediaMIDI,
                                      m_MIDINoteOnMsgBuf, 3);
iReturn = IMEDIA_Play(m_pIMediaMIDI);
```

9.5.3.2 Using MIDI Messaging in combination with MIDI files

MIDI messages can be used concurrently with the playback of MIDI files. If there are multiple MIDI files, only the playback of the first MIDI file will be affected by the MIDI commands as illustrated in Figure 9-10. In this case, the MIDI command will be sent asynchronously to the MIDI synthesizer for the first MIDI file.

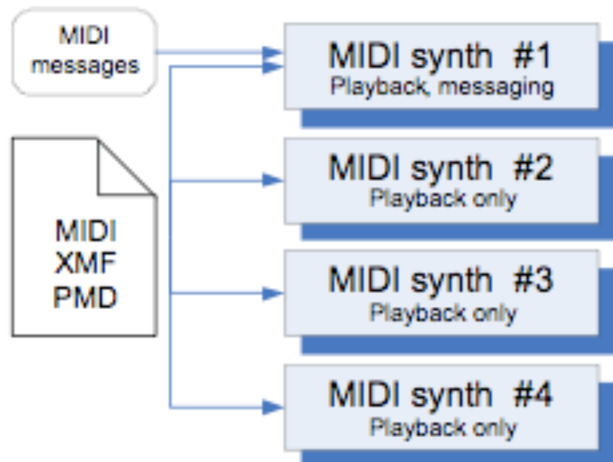


Figure 9-10 MIDI synthetizers within CMX

Example:

The following code illustrates the usage of MIDI commands concurrently with playback of a MIDI file:

```
// If MIDI out messages are going to affect an existing
// MIDI file, the file object must be created.
iReturn = ISHELL_CreateInstance(m_pIShell, AEECLSID_MEDIAMIDI,
                                (void **)&m_pIMediaMIDIFile);

// Set the MIDI file as data to the IMedia object
AEEMediaData   MediaData;
MediaData.clsData = MMD_FILE_NAME;
MediaData.pData = pFileName;
MediaData.dwSize = 0;
iReturn = IMEDIA_SetMediaData(m_pIMediaMIDIFile, &MediaData);

// Must start the file playing in order to modify it with
// the MIDI out messages.
iReturn = IMEDIA_Play(m_pIMediaMIDIFile);


// The MIDI out message must be created using
// AEECLSID_MEDIAMIDIOUTMSG.  If AEECLSID_MEDIAMIDI is
// used the call to IMEDIAMIDIOUTMSG_SetMIDIMsg will return
// EUNSUPPORTED.
iReturn = ISHELL_CreateInstance(m_pIShell, AEECLSID_MEDIAMIDIOUTMSG,
                                (void **)&m_pIMediaMIDIOut);

// You MUST call IMEDIA_SetMediaData or you cannot send
```

```
// MIDI messages
AEEMediaData   MediaData;
m_MIDIMsgBuf[0] = 0xB0;
m_MIDIMsgBuf[1] = 0x07;
m_MIDIMsgBuf[2] = 0x7F;
MediaData.clsData = MMD_BUFFER;
MediaData.pData = m_MIDIMsgBuf;
MediaData.dwSize = 3;

iReturn = IMEDIA_SetMediaData(m_pIMediaMIDIOut, &MediaData);

// Can only have one outstanding MIDI out message.  If another
// is sent before the first is finished EBADSTATE is returned.
// Need to RegisterNotify in order to get status messages
// and make sure this doesn't happen.
iReturn = IMEDIA_RegisterNotify(m_pIMediaMIDIOut,
                                CMIDIMsg::MediaNotify, this);

// The first time MIDI message is being ignored
// To get around this, call play immediately after calling SetMediaData
// on dummy data
iReturn = IMEDIA_Play(m_pIMediaMIDIOut);

// Now, to send a MIDI command (like turning on
// a note)  do the following.  This note will be on top
// of any playing MIDI file.
m_MIDINoteOnMsgBuf[0] = 0x90;
m_MIDINoteOnMsgBuf[1] = 0x3C;
m_MIDINoteOnMsgBuf[2] = 0x7F;
iReturn = IMEDIAMIDIOUTMSG_SetMIDIMsg(m_pIMediaMIDIOut,
                                      m_MIDINoteOnMsgBuf,3);
iReturn = IMEDIA_Play(m_pIMediaMIDIOut);

// Can mute channel 0 in the playing MIDI file
// by doing the following
m_MIDINoteOnMsgBuf[0] = 0xB0;
m_MIDINoteOnMsgBuf[1] = 0x07;
m_MIDINoteOnMsgBuf[2] = 0x00;
iReturn = IMEDIAMIDIOUTMSG_SetMIDIMsg(m_pIMediaMIDIOut,
                                      m_MIDINoteOnMsgBuf, 3);
iReturn = IMEDIA_Play(m_pIMediaMIDIOut);
```

9.5.4 QAudioFX – 3D Audio

The QAudioFX feature includes the capability to control listener and sound source settings. These include accessing the listener position and orientation, and the sound source position, volume, and rolloff. In the next two sections, the listener and sound source settings are described in detail.

QAudioFX is not available for this first release of Zeebo wireless gaming platform but it is in the roadmap for 2009.

All of the BREW APIs for listener and sound source settings includes a dwDuration parameter. This parameter represents the time, in milliseconds, over which the change in

playback occurs. This parameter is not currently supported and defaults to 0. Do not confuse the 3D flag with the feature 3D. Positions are all set using Cartesian coordinates.

9.5.4.1 Enabling positional audio and rolloff effects

To hear any positional audio or rolloff effects, the application must explicitly enable 3D on all objects for which the effects are desired, create an audio environment for which 3D is enabled, and attach all of the audio objects to the environment. The settings for the listener, the environment, or the sound source may be updated at any time; however, the effect of the updates will only be apparent when the requirements for positional audio are met.

To enable positional audio effects and rolloff, the application must set the mode for each audio object. The BREW API used for this is:

```
IMEDIAAUDIO3D_SetMode(IMediaAudio3D *me,
                      int nMode)
```

The value of nMode is summarized in Table 9-1.

Table 9-1 Value of nMode

Value of nMode	Description
0	Positional effects off
1	Absolute positional effects on

All positional audio effects are relative to the origin and not the listener. It should be noted here that the flag to enable positional effects is the same flag that is set for Doppler effects. There is no flag for explicitly enabling reverb. This will be covered in a subsequent section.

Next, the application must enable positional effects on the environment using the BREW API:

```
IMEDIAAUDIOENV_Enable(IMediaAudioEnv *me,
                      uint32 *pdwNew,
                      uint32 *pdwOld)
```

The value of the pdwNew parameter settings is summarized in .

Table 9-2 Value of pdwNew

Value of pdwNew	Description
0	No effects enabled in the environment
1	Positional effects enabled including Doppler
2	Reverb effects enabled
-1	All effects enabled

It is important to note that for enabling environment effects, the pdwOld parameter must be set according to all desired effects. For example, by first calling this API with only

positional enabled and then again with reverb enabled, this does not equate to calling the API with all effects enabled. Only the last call to this API determines what is enabled or not.

Lastly, the audio object must be attached to this environment. Only one environment may be created and used at a time. The BREW API for this is as follows:

```
int IMEDIAAUDIOENV_AddMedia(
                  IMediaAudioEnv *pIMediaAudioEnv,
                  IMedia *pIMedia)
```

9.5.4.2 Listener and environment settings

The listener is an inherent part of the audio environment. Current implementation supports the following listener and environment BREW APIs for positional audio:

- `IMEDIAAUDIOENV_SetListenerPosition();`
- `IMEDIAAUDIOENV_GetListenerPosition();`
- `IMEDIAAUDIOENV_SetListenerOrientation();`
- `IMEDIAAUDIOENV_GetListenerOrientation();`

9.5.4.2.1 Listener Position

The listener position can be retrieved and set by the application. Currently, Cartesian coordinates for the listener position are supported. To set the listener position, the application may issue a call to:

```
IMEDIAAUDIOENV_SetListenerPosition
    (
    IMediaAudioEnv *pIMediaAudioEnv,
    AEEVector *pPos,
    uint32 dwDuration
    )
```

9.5.4.2.2 Listener Orientation

The listener orientation can also be controlled by the application. The listener orientation is used to set the direction that the listener is facing. The BREW APIs used to set the listener orientation is as follows:

```
IMEDIAAUDIOENV_SetListenerOrientation
    (
    IMediaAudioEnv *pIMediaAudioEnv,
    AEEMediaOrientation *pmo,
    uint32 dwDuration
    )
```

9.5.4.3 Sound Source Settings

The sound source settings affect the position, volume, and rolloff of a media object.

- `IMEDIAAUDIO3D_SetPosition();`
- `IMEDIAAUDIO3D_GetPosition();`
- `IMEDIAAUDIO3D_SetRollOff();`
- `IMEDIAAUDIO3D_GetRollOff();`
- `IMEDIAAUDIO3D_SetVolume();`
- `IMEDIAAUDIO3D_GetVolume();`

With QAudioFX 1.0, all API calls for sound source settings must occur after object playback has started. The application must first play the object, wait for the MM_STATUS_START, and then set the position/rolloff/volume of the object. Trying to set these parameters prior to starting playback will result in error. This restriction does not apply with QAudioFX 1.1 and later.

9.5.4.3.1 Sound Source Position

The sound source position can be set using the BREW API:

```
IMEDIAAUDIO3D_SetPosition
    (
    IMediaAudio3D *pIMediaAudio3D,
    AEEVector *pPos,
    uint32 dwDuration
    )
```

There may be a slight delay after setting the position before the new value can be read back via IMEDIAAUDIO3D_GetPosition.

9.5.4.3.2 Sound Source Rolloff

The sound source rolloff specifies how the volume will change with distance. Sound sources that are closer than the minimum distance are played at maximum volume; sounds further than the maximum distance remain at constant volume or are muted, and sounds in between are attenuated exponentially with the rolloff factor as distance increases. All distances are measured in millimeters. The mute-after-max flag is set if it is desired that sound objects positioned farther than the maximum distance be muted. The rate of attenuation with distance can be modified with AEEMediaRollOff.

The BREW API for setting the source rolloff parameters is:

```
IMEDIAAUDIO3D_SetRollOff
    (
    IMediaAudio3D *pIMediaAudio3D,
    AEEMediaRollOff *prf,
    uint32 dwDuration
    )
```

The structure for AEEMediaRollOff is:

```
typedef struct AEEMediaRollOff {
    int32   nMinDistance;    //[Mandatory] Millimeters.
    int32   nMaxDistance;    //[Mandatory] Millimeters.
    int32   nRollOffactor;  //[Mandatory] In thousandths.
    boolean bMuteAfterMax;   //[Mandatory]
}
```

A rolloff factor of 1000 (1.0) is normal rolloff and 0 is no rolloff.

9.5.4.3.3 Sound Source Volume

The application may also control the volume of each audio object using the following BREW API:

```
IMEDIAAUDIO3D_SetVolume
    (
    IMediaAudio3D *pIMediaAudio3D,
    int32 *pnVolume,
    uint32 dwDuration
    )
```

9.5.4.4 Reverberation Effects

Reverberation effects can be used in conjunction with the playback of an audio object. Reverberation is defined as the effect when multiple reflections of sound source off of a surface are summed together. The reverberation is controlled by the physical surroundings that the audio object is in.

9.5.4.4.1 Enabling and disabling reverberation effects

The current implementation supports the enabling and disabling of reverb effects on an environment. Since there are no explicit BREW APIs to enable or disable reverberation effects, the application will have to manipulate the reverb levels for each object for which reverb effects are desired.

To enable reverb effects for the audio environment, use the following BREW API:

```
int IMEDIAAUDIOENV_Enable(
               IMediaAudioEnv *pIMediaAudioEnv,
               uint32 *pdwNew,
               uint32 *pdwOld)
```

With pdwNew set to 2 (enable reverb) or -1 (enable all QAudioFX).

Since all newly created objects have a reverb level set to -9600, which is the minimum reverb volume, it will be necessary to update the reverb level of the audio object using the following BREW API:

```
int IMEDIAAUDIOFX_SetReverbGain(
                IMediaAudioFX *me,
                int32 nGain)
```

As with positional audio, the object must be attached to the environment. See section 9.5.4.1 for details about the BREW API for adding objects to the environment.

9.5.4.4.2 Setting Reverb parameters for the environment

There are several parameters that can be used to change the reverb effects. These include reverb preset, reverb gain, decay time, and damping factor. Setting the room size is not currently supported. To start, a reverb preset as described in section 9.3.3 must be selected using the BREW API:

```
int IMEDIAAUDIOFX_SetReverbPreset(
                IMediaAudioFX *pIMediaAudioFX,
                int32 nPreset)
```

The reverb gain, damping factor, and decay time for the environment can be updated using the following BREW APIs:

```
int IMEDIAAUDIOFX_SetReverbGain
    (
    IMediaAudioFX *pIMediaAudioFX,
    int32 nGain
    )
```

The reverb gain, nGain, is specified within a range from -9600 to 0 millibels. Since by default the reverb gain for the environment, as well as the audio objects, is -9600, it will be necessary to call the above API to hear any reverb if a preset is not selected. If a preset is selected, then the associated reverb gain for that particular preset will be used for the environment reverb gain.

```
int IMEDIAAUDIOFX_SetReverbDampingFactor
    (
    IMediaAudioFX *pIMediaAudioFX,
    uint32 dwDamping
    )
```

As with reverb gain, setting the reverb damping factor, dwDamping, will update the current value of the damping factor. Using a reverb preset will automatically initialize this value. This parameter ranges from 0 to 2000, where 1000 applies equal weighting to both low and high frequencies. A low damping factor means that high frequencies decay very quickly, whereas a high damping factor implies the reverse.

```
int IMEDIAAUDIOFX_SetReverbDecayTime
    (
    IMediaAudioFX *pIMediaAudioFX,
    uint32 dwTimeMS
    )
```

Again, this parameter is initialized automatically by choosing a reverb preset. The decay time is measured in milliseconds. Decay time is a measure of how "live" the room is. The larger the decay time, the longer the echoes will sound. Recall that any updates to these parameters will only affect those objects that are currently attached to the environment.

9.5.4.4.3 Setting Reverb parameters for audio objects

Currently, it is only possible to update the reverb gain and decay time on a per-object basis. The BREW APIs for this are the same as those for the environment.

9.5.4.5 Doppler Effects

9.5.4.5.1 Enabling Doppler Effects

The same procedures and flag that are used for enabling positional audio are also used to enable Doppler effects. To hear noticeable Doppler effects, the listener and the sound source must have a relative velocity. A noticeable frequency shift should occur when the relative velocity is large. The parameters that are involved in calculating this shift are the object and listener positions and velocities. So to hear a changing pitch, the application must also move the object and/or listener positions. Setting the velocity does not automatically update the position of either the object or the listener.

The current implementation supports Doppler rates in the range 0.25 to 4.0. Doppler rate is defined in the following equation (c = speed of sound, V = listener velocity, and V = object velocity):

$$DopplerRate = \frac{c + V_{listener}}{c + V}$$

9.5.4.5.2 Setting Doppler parameters for the listener and environment

Setting and getting the velocity of the listener is supported in current implementation of QAudioFX. To set the velocity of the listener, use the following BREW API:

```
int IMEDIAAUDIOENV_SetListenerVelocity
    (
    IMediaAudioEnv *pIMediaAudioEnv,
    AEEVector *pVelocity
   )
```

Where pVelocity is a vector describing listener velocity including the x, y, and z fields and is measured in millimeters/second.

9.5.4.5.3 Setting Doppler parameters for audio objects

Next, the application can set a velocity for the audio object. A similar API as that used for the listener is used to set velocity:

```
int IMEDIAAUDIO3D_SetVelocity
    (
    IMediaAudio3D *pIMediaAudio3D,
    AEEVector *pVelocity,
    uint32 dwDuration
    )
```

9.5.4.6 QAudioFX – Putting it all together

9.5.4.6.1 Disabling specific features

To selectively disable certain effects and keep others, the application can disable the per-object flag for that particular effect if a flag is available, i.e., positional audio, or it can simply zero-out a setting for that effect, i.e., reverb effects. If it desired that the audio object play back as background music without any QAudioFX effects at all, the object can simply be removed from the environment. Section 9.5.4.6.2 explores particular use case scenarios to illustrate the behavior of positional audio, rolloff, reverb, and Doppler effects.

9.5.4.6.2 QAudioFX parameters and recommendations

This section presents a summary of QAudioFX parameters ranges and recommendations. It is recommended to use ADPCM or PCM input audio that is sampled at 11.025 kHz or higher for good audio fidelity when using QAudioFX.

Table 6-4 delineates QAudioFX parameters, ranges, and special notes. As is evident in the table, at times the engine clips the specified parameter because of internal limitations or interparameter dependencies.

Table 9-3 QAudioFX parameters and ranges

	Parameter		Default parameter value	Parameter range	Special notes
Positional	Listener position Cartesian	X	0 mm	Signed 32-bit integer	
		Y	0 mm	Signed 32-bit integer	
		Z	0 mm	Signed 32-bit integer	
	Listener orientation up	X	0 mm	Signed 32-bit integer	
		Y	1 mm	Signed 32-bit integer	
		Z	0 mm	Signed 32-bit integer	
	Listener orientation forward	X	0 mm	Signed 32-bit integer	
		Y	0 mm	Signed 32-bit integer	
		Z	-1 mm	Signed 32-bit integer	
	Object Position Cartesian	X	0 mm	Signed 32-bit integer	
		Y	0 mm	Signed 32-bit integer	
		Z	0 mm	Signed 32-bit integer	
	Volume		0 mm	-9600~0	Volume is clamped to range
	Min Distance		1000 mm	Signed 32-bit integer	

	Parameter		Default parameter value	Parameter range	Special notes
	Max Distance		2147483647 mm	Signed 32-bit integer	
	Rollof factor		1000 (1.0) thousandths	Signed 32-bit integer	
	Mute-after-max		0 (hold after max)	0 or 1	
Reverb	Preset		None	None, room, bathroom, concert hall, cave, arena, forest, city, mountains, underwater, auditorium, alley, hallway, hangar, living room, small room, medium room, large room, medium hall, large hall, plate	
	Decay time		0x000F(15)	Unsigned 31 bit integer	Range will be clipped to [15, 32767]
	Object reverb gain		-9600 mB (silence)	-9600~0	Range clipped to -9600~0
	Damping Factor		0x10000(Q16 one)	Unsigned 31-bit integer	Internal value is dependent on specified reverb decay time
Doppler	Linear velocity Cartesian	X	0 mm/sec	Signed 32-bit integer	Doppler rate must be [0.25, 4.0] See Doppler rate equation in section 9.5.4.5.1
		Y	0 mm/sec	Signed 32-bit integer	
		Z	0 mm/sec	Signed 32-bit integer	
	Object velocity Cartesian	X	0 mm/sec	Signed 32-bit integer	
		Y	0 mm/sec	Signed 32-bit integer	
		Z	0 mm/sec	Signed 32-bit integer	

9.5.4.6.3 Use case scenario

This section contains code samples that illustrate how to achieve positional audio, reverb, and Doppler effects.

Example:

The following code plays an audio object at position (0, 1000, 0) with rolloff, reverberation effects with the CAVE preset, and Doppler effects with object velocity set to 0.5 m/s in the x direction.

```
//-------------------------------------------------------------------
boolean InitAudio(TSimpleAudioReverb* pMe)
//-------------------------------------------------------------------
{
    int             iReturn = SUCCESS;
    char            *pFileName = "testing123.wav";
    AEECLSID        WhichCLSID = AEECLSID_MEDIAPCM;
    AEEMediaData    MediaData;
    uint32          dwNewValue = MM_AENV_ENABLE_3D | MM_AENV_ENABLE_REVERB;
    uint32          dwOldValue;
    IMediaAudio3D   *pAudio3D = NULL;
```

```
AEEVector       PositionVector;
AEEVector       VelocityVector;
AEEMediaRollOff RollOff;
int             iMediaState;
boolean         fIsChanging;

if(pMe == NULL)
    return FALSE;

// First create the IMedia object
iReturn = ISHELL_CreateInstance(pMe->a.m_pIShell, WhichCLSID,
                                (void **)&pMe->pIMediaObject);
if(iReturn != SUCCESS)
{
    DBGPRINTF("Error creating media: 0x%X", iReturn);
    pMe->pIMediaObject = NULL;
    return FALSE;
}

// Then set media data
MediaData.clsData = MMD_FILE_NAME;
MediaData.pData = pFileName;
MediaData.dwSize = 0;

iReturn = IMEDIA_SetMediaData(pMe->pIMediaObject, &MediaData);
if(iReturn != SUCCESS)
{
    DBGPRINTF("Error setting media data: 0x%X", iReturn);
    return FALSE;
}

// Set the Media notification routine
iReturn = IMEDIA_RegisterNotify(pMe->pIMediaObject, MediaNotify, pMe);
if(iReturn != SUCCESS)
{
    DBGPRINTF("Error setting media notify callback function: 0x%X",
            iReturn);
    return FALSE;
}

// Now create the environment
iReturn = ISHELL_CreateInstance(pMe->a.m_pIShell,
             AEECLSID_MEDIAAUDIOENV, (void **)&pMe->pIMediaEnv);
if(iReturn != SUCCESS)
{
    DBGPRINTF("Error creating media environment: 0x%X", iReturn);
    pMe->pIMediaEnv = NULL;
    return FALSE;
}

// Enable 3D and reverb on the environment
iReturn = IMEDIAAUDIOENV_Enable(pMe->pIMediaEnv, &dwNewValue,
                                 &dwOldValue);
if(iReturn != SUCCESS)
{
    DBGPRINTF("Error enabling 3D audio in environment: 0x%X", iReturn);
    return FALSE;
}
```

```
// Get the FX interface to the environment and set reverb preset
iReturn = IMEDIAAUDIOENV_QueryInterface(pMe->m_pIMediaEnv,
             AEEIID_MEDIAAUDIOFX, (void **)&pAudioFX);
if(iReturn != SUCCESS)
{
    pMe->DisplayError(IDS_ERR_GET_ENV_AUDIOFX, iReturn);
    break;
}

iReturn = IMEDIAAUDIOFX_SetReverbPreset(pAudioFX, CAVE);
if(iReturn != SUCCESS)
{
   IMEDIAAUDIOFX_Release(pAudioFX);
   pMe->DisplayError(IDS_ERR_SET_ENV_PRESET, iReturn);
   break;
}

// Add media object to environment
iReturn = IMEDIAAUDIOENV_AddMedia(pMe->pIMediaEnv, pMe->pIMediaObject);
if(iReturn != SUCCESS)
{
    DBGPRINTF("Error adding media to audio environment: 0x%X",
                iReturn);
    return FALSE;
}

// Must enable 3D on the object also.  In order to do this, we need
// the 3D interface.
iReturn = IMEDIA_QueryInterface(pMe->pIMediaObject,
                                 AEECLSID_MEDIAAUDIO3D,
                                 (void **)&pAudio3D);
if(iReturn != SUCCESS)
{
    DBGPRINTF("Error getting 3D interface to media: 0x%X", iReturn);
    return FALSE;
}

// Got the 3D interface, enable 3D on the object
iReturn = IMEDIAAUDIO3D_SetMode(pAudio3D, MM_A3D_MODE_NORMAL);
if(iReturn != SUCCESS)
{
    DBGPRINTF("Error setting 3D mode for media: 0x%X", iReturn);
    IMEDIAAUDIO3D_Release(pAudio3D);
    return FALSE;
}

// Set the position, velocity, and rolloff values before starting play
PositionVector.x = 0;
PositionVector.y = 1000;
PositionVector.z = 0;
iReturn = IMEDIAAUDIO3D_SetPosition(pAudio3D, &PositionVector, 0);
if(iReturn != SUCCESS)
{
    DBGPRINTF("Error setting 3D position for media: 0x%X", iReturn);
    IMEDIAAUDIO3D_Release(pAudio3D);
    return FALSE;
}
```


Zeebo Confidential and Proprietary

```
VelocityVector.x = 500;
VelocityVector.y = 0;
VelocityVector.z = 0;
iReturn = IMEDIAAUDIO3D_SetVelocity(pAudio3D, &VelocityVector, 0);
if(iReturn != SUCCESS)
{
    DBGPRINTF("Error setting 3D velocity for media: 0x%X", iReturn);
    IMEDIAAUDIO3D_Release(pAudio3D);
    return FALSE;
}


RollOff.nMinDistance = ROLLOFF_MIN_DIST;
RollOff.nMaxDistance = ROLLOFF_MAX_DIST;
RollOff.nRollOffactor = ROLLOFF_FACTOR;
RollOff.bMuteAfterMax = FALSE;
iReturn = IMEDIAAUDIO3D_SetRollOff(pAudio3D, &RollOff, 0);
if(iReturn != SUCCESS)
{
    DBGPRINTF("Error setting 3D rolloff for media: 0x%X", iReturn);
    IMEDIAAUDIO3D_Release(pAudio3D);
    return FALSE;
}

// No longer need the 3D interface to the media
IMEDIAAUDIO3D_Release(pAudio3D);
pAudio3D = NULL;


// Set the repeat flag to 0, for infinite looping
iReturn = IMEDIA_SetMediaParm(pMe->pIMediaObject, MM_PARM_PLAY_REPEAT,
                              0, 0);
if(iReturn != SUCCESS)
{
    // This is not a fatal error since we will just restart it when
    // we get MM_STATUS_DONE.
    DBGPRINTF("Error setting repeat count for media: 0x%X", iReturn);
}
// Finally, start the play
iMediaState = IMEDIA_GetState(pMe->pIMediaObject, &fIsChanging);
if(iMediaState != MM_STATE_PLAY && !fIsChanging)
{
    iReturn = IMEDIA_Play(pMe->pIMediaObject);
    if(iReturn != SUCCESS)
    {
        DBGPRINTF("Error starting play: 0x%X", iReturn);
        return FALSE;
    }
}
else
{
    DBGPRINTF("Error starting play: Media in unknown state");
    return FALSE;
}


return TRUE;
```


Zeebo Confidential and Proprietary

```
}
```

To disable any of the effects above, the application can simply disable per-object flags, set parameters such as reverb level to 0, and/or remove the object from the environment if needed.

10 Zeebo.lib System Library

The Zeebo system library provides a high-level API for dealing with basic functionalities present in the console. The API is built on top of the standard BREW API and is distributed to standardize and make the game development easier for the platform. A header file with all the function prototypes and a compiled version of the library to be linked against your game are provided in the SDK.

Zeebo library is represented as an interface, named IZeebo that uses similar BREW initialization and releasing methods. Developers must call Zeebo_Initialize(pShell) prior to use any functionality present on the library and a Zeebo_Free(pZeebo) must be called while exiting the game, releasing all resources related to Zeebo.lib. A Zeebo event handler method, named Zeebo_HandleEvent(pZeebo, evt, wp, dwp) must be called inside the game's event loop to handle Zeebo's virtual keyboard specific events. A detailed documentation of each function call and parameters is available in the header file ZeeboLib.h provided with the SDK.

A set of features such as suspending and resuming the game, functions for handling the virtual keyboard and gamepad detection are described in the next sections. The suspend and resume events and the gamepad detection features are mandatory for all Zeebo games.

10.1 Suspend and Resume

The suspend and resume events generated while pressing the Home button on Zeebo gamepad will be handled by a background application running at the same time the games are played.

The default behavior is, if the Home button has been pressed for a period longer than 3 seconds, a popup screen asks whether the user wants to return to the Stage or continue to play the game. A suspend event is then sent to the game. Returning to the Stage will generate a close game event (IShellCloseApplet()) while returning to game will generate a resume event.

However, if the Home button is pressed for a period shorter than 3 seconds, a standard keypress event – AVK_CLR - is directed to the game and interpreted as an in-game pause.

10.2 Virtual Keyboard

A virtual keyboard is available for input text based information into the game. A virtual keyboard appears over the game screen. Use Zeebo_CreateKeyboard() to invoke and show the virtual keyboard on the lower half of the display. The virtual keyboard translates the user selections on the screen to EVT_CHAR events which can be passed to a text control or

to the main game event handler. Calling Zeebo_CloseKeyboard() will close the virtual keyboard and a EVT_DIALOG_END event will be sent to the game.

10.3 Controller Discovery and Removal

Developers are able to determine how many controllers are plugged and indentify which one is “the Player 1” and “the Player 2”. Calling Zeebo_DeviceDetectAndSelect() delivers to the game the proper object handler for the connected IHID device, listening to a button press and sending an EVT_ZEEBO_DEVICESELECT event to the game.

The method Zeebo_GetDetectedDevice() returns a IHIDevice handle received from the EVT_ZEEBO_DEVICESELECT event. The caller should be responsible for releasing the returned IHIDDevice. Developers also should be able to cancel the device detection process by calling Zeebo_DeviceDetectCancel().

Games are notified on the main event loop with the EVT_ZEEBO_DEVICEREMOVE event when a gamepad is removed from the USB port. Plugging a new gamepad, EVT_ZEEBO_DEVICEADD event is sent to the game event loop.

The controller discovery and removal can also be determined by using the BREW IHID extensions function calls directly. For more details on how to use the IHID extension directly and how to handle events and notifications from connected gamepads, See section 6.

11 Additional Information and Requirements

11.1 Compiling Zeebo Games

Zeebo will provide a sample application exploiting the key features present in the SDK to illustrate the usage of the high level API of the Zeebo.lib.

Developers should use ARM Real View Developer Suite 2.2 (RVDS 2.2) or higher to achieve the best performance on ARM code generated for the processors present in the console. A standard makefile with the key optimizations options is also included in the SDK.

11.2 Tuning for TV-Out

Remember that the Zeebo is targeted for developing countries which currently have an estimated installed base of more than 90% older color CRT TVs. We recommend buying a CRT TV for testing; they can still be found at retailers such as Best Buy, Fnac, Sanborns or Ebay.

Games for the Zeebo console must be developed targeting a VGA (640x480) screen size, regardless the TV resolution. Developers should keep in mind that older TVs based on cathode ray tubes are rounded and this could distort or cut some pieces of the final image shown on it. Applications should reserve a "safe area" of four to eight pixels and display no content in that area on the top, bottom, left, and right borders of all application screens.

The MDP processor will take care of different TV resolutions, resampling the final image on the TV screen. Please refer to section 8.3 for more details on how you can optimize performance on Zeebo rescaling the video output.

11.3 Saving data to console

All data generated by games and is supposed to reside or be saved inside (e.g. user configuration, save game, etc…) the console for further uses must be saved in a separate folder, named **udata**, located inside the main game folder.

File system configuration must follow the next pattern:

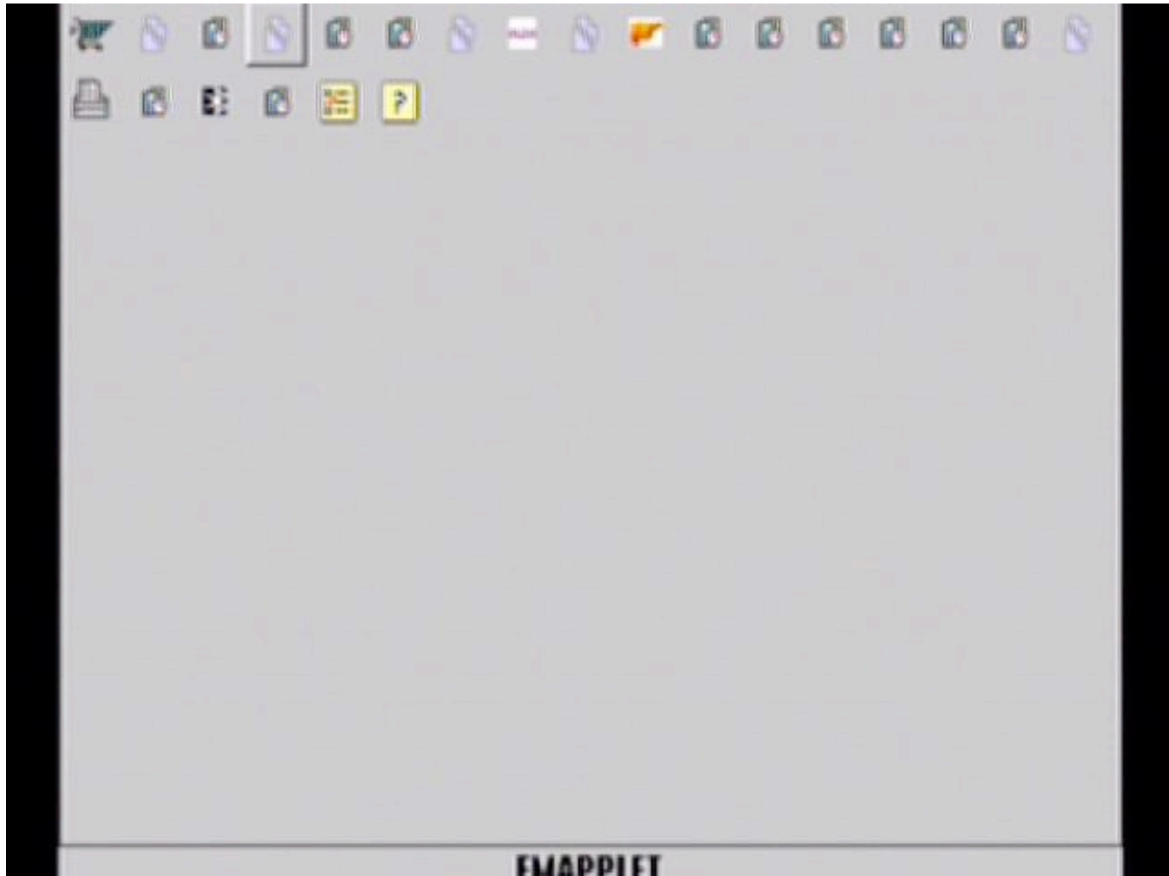
/mod/yourgamefolder/ ← main game folder

/mod/yourgamefolder/udata/ ← game generated data must be stored here.

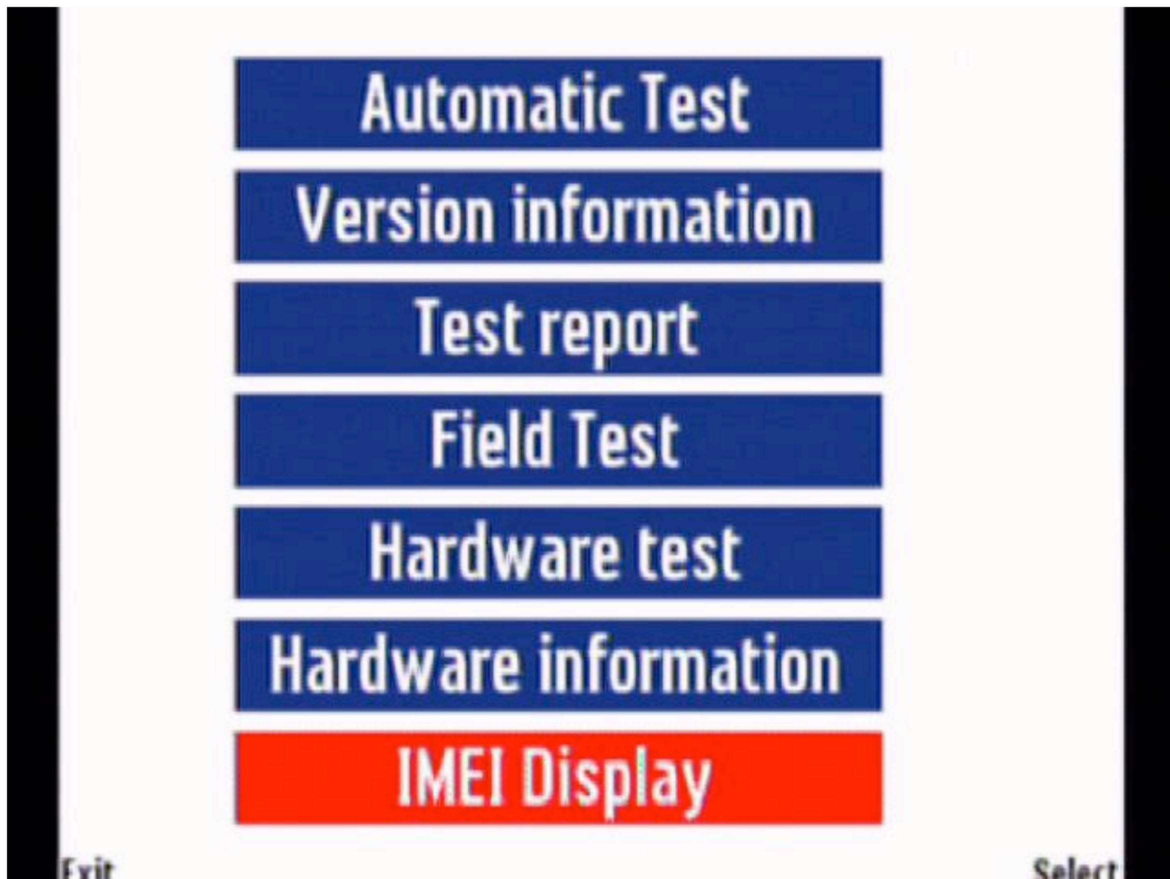
11.4 Using BREW AppLoader

Steps for using BREW AppLoader with Zeebo:

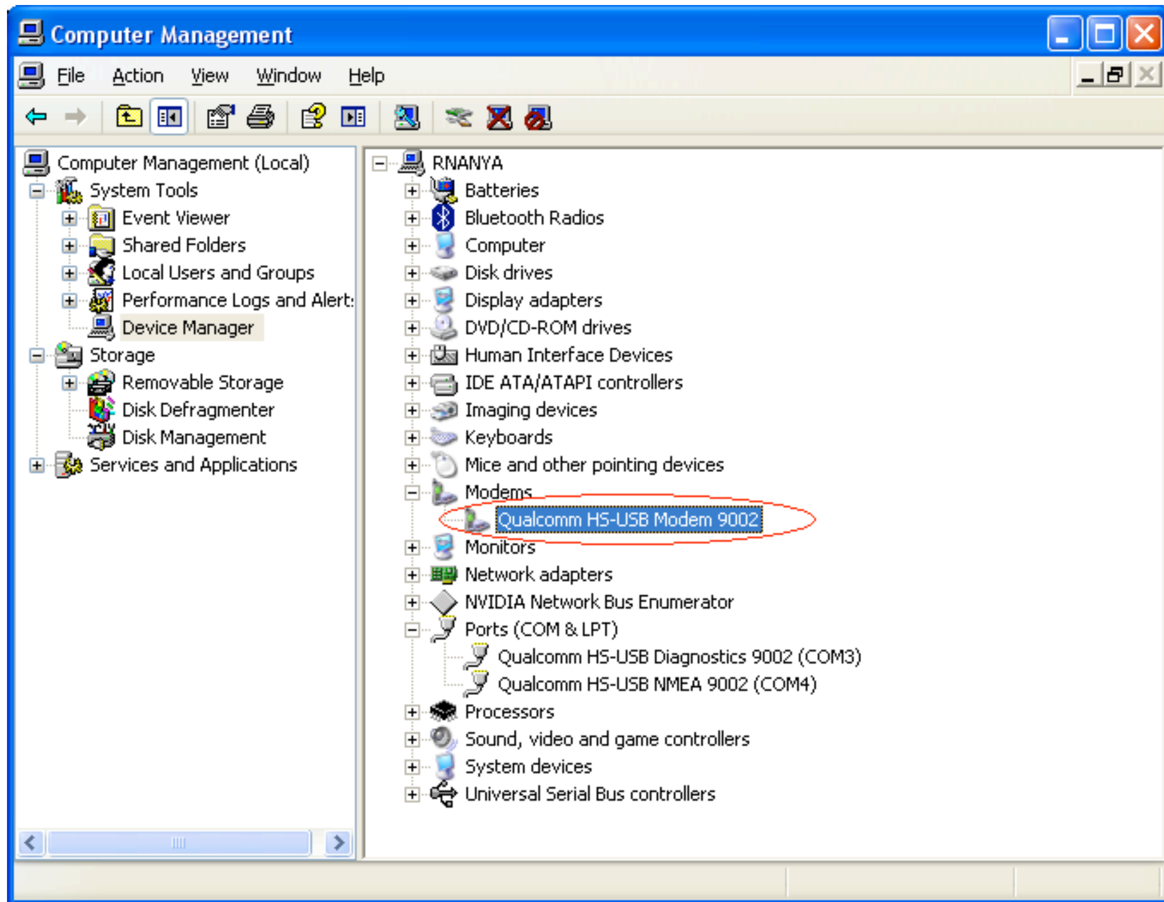
1. Check Zeebo console's IMEI number
   - In the Main Menu screen (item list), start EMAPPLET application.

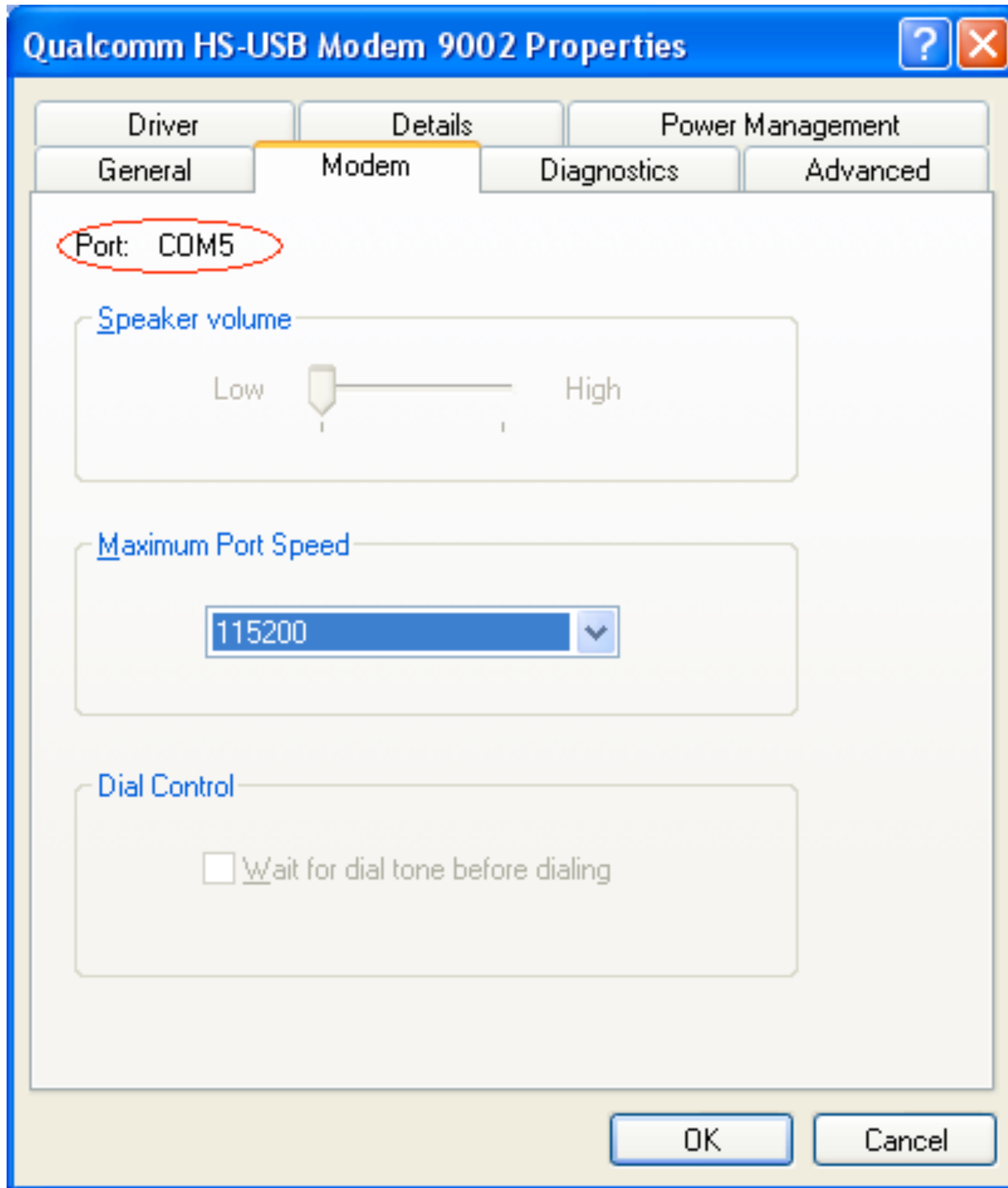


   - Select IMEI Display and you will be able to see console's IMEI number.

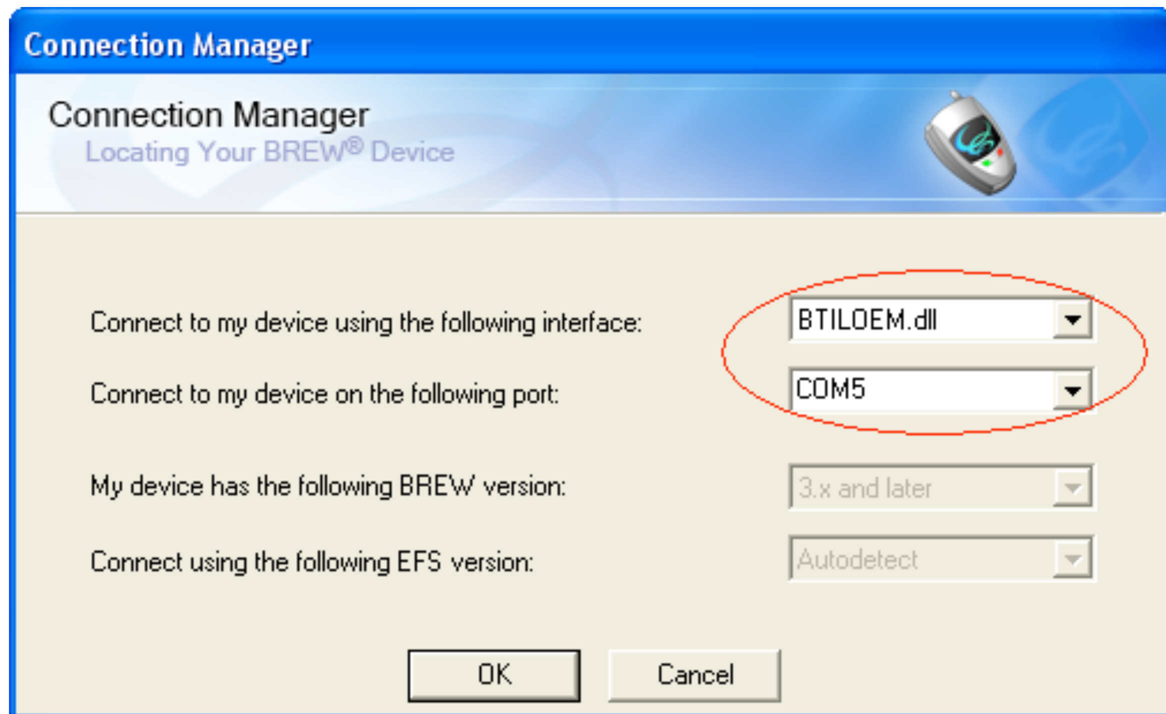


2. Generate the proper test signature file (.sig) for the console. Test signature files can be generated in Qualcomm's BREW developers extranet.
3. Make sure you have a SIM card installed in your console.
4. Copy the test signature file to BTIL Development Kit folder (usually C:\Program Files\Common Files\Qualcomm\BTIL Development Kit\Host\sig).
5. Connect Zeebo to PC and check Qualcomm Modem COM port number. Ensure the console is in USB Download mode – see 11.6 for more details.


Computer Management
File Action View Window Help
Computer Management (Local)
System Tools
Event Viewer
Shared Folders
Local Users and Groups
Performance Logs and Alert:
Device Manager
Storage
Removable Storage
Disk Defragmenter
Disk Management
Services and Applications
RNANYA
Batteries
Bluetooth Radios
Computer
Disk drives
Display adapters
DVD/CD-ROM drives
Human Interface Devices
IDE ATA/ATAPI controllers
Imaging devices
Keyboards
Mice and other pointing devices
Modems
Qualcomm HS-USB Modem 9002
Monitors
Network adapters
NVIDIA Network Bus Enumerator
Ports (COM & LPT)
Qualcomm HS-USB Diagnostics 9002 (COM3)
Qualcomm HS-USB NMEA 9002 (COM4)
Processors
Sound, video and game controllers
System devices
Universal Serial Bus controllers



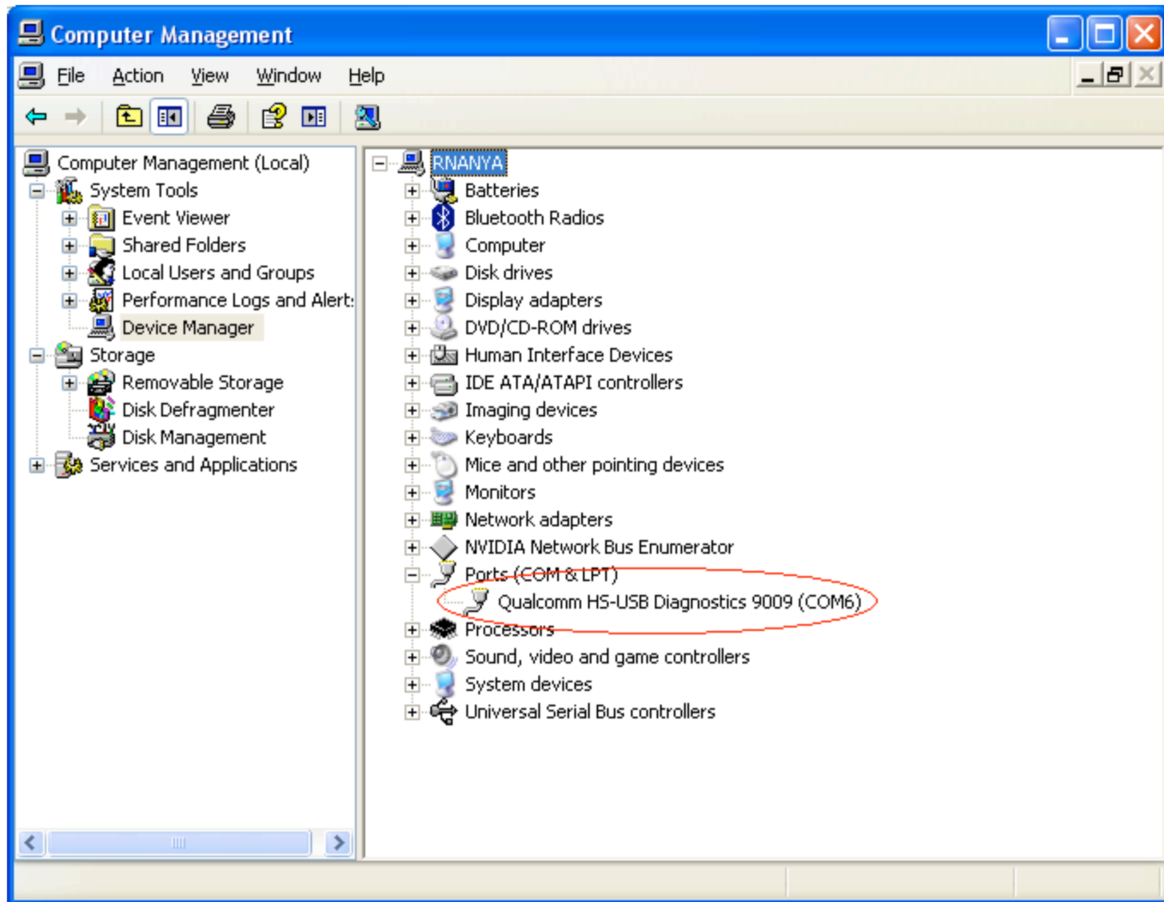
6. Start BREW AppLoader, selecting the correct COM port (see item 4) and BTILOEM.dll.



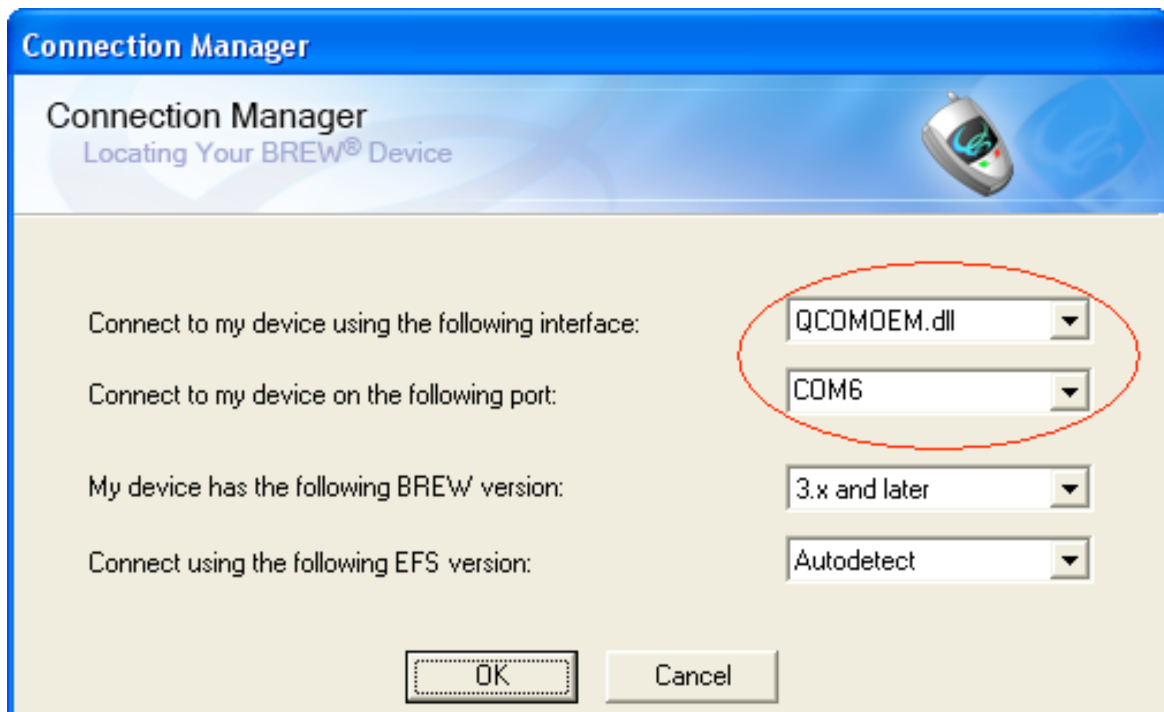
11.5 Using BREW Logger

Steps for using BREW Logger with Zeebo:

1. Connect Zeebo to PC and check Qualcomm HS-USB Diganostics 9009 COM port number. Ensure the console is in USB Trace mode – see 11.6 for more details.



2. Start BREW Logger, setting the correct COM port and selecting QCOMOEM.dll as the connection interface.



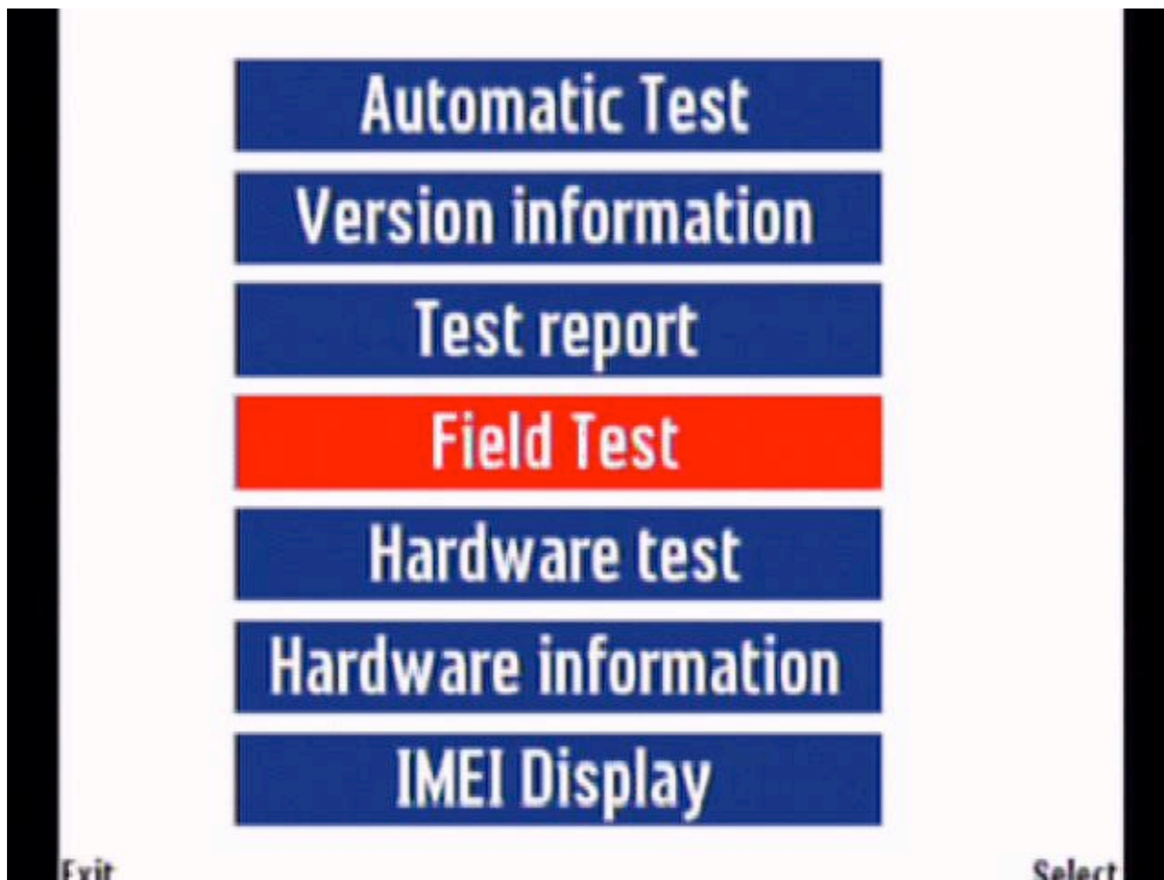
11.6 Understanding USB Download and Trace mode

USB Download mode is used for uploading and downloading files/contents between Zeebo and PC (while using BREW AppLoader). On the other hand, USB trace mode is mainly used for logging and debugging applications running on Zeebo (while using BREW Logger or Adreno Profiler).

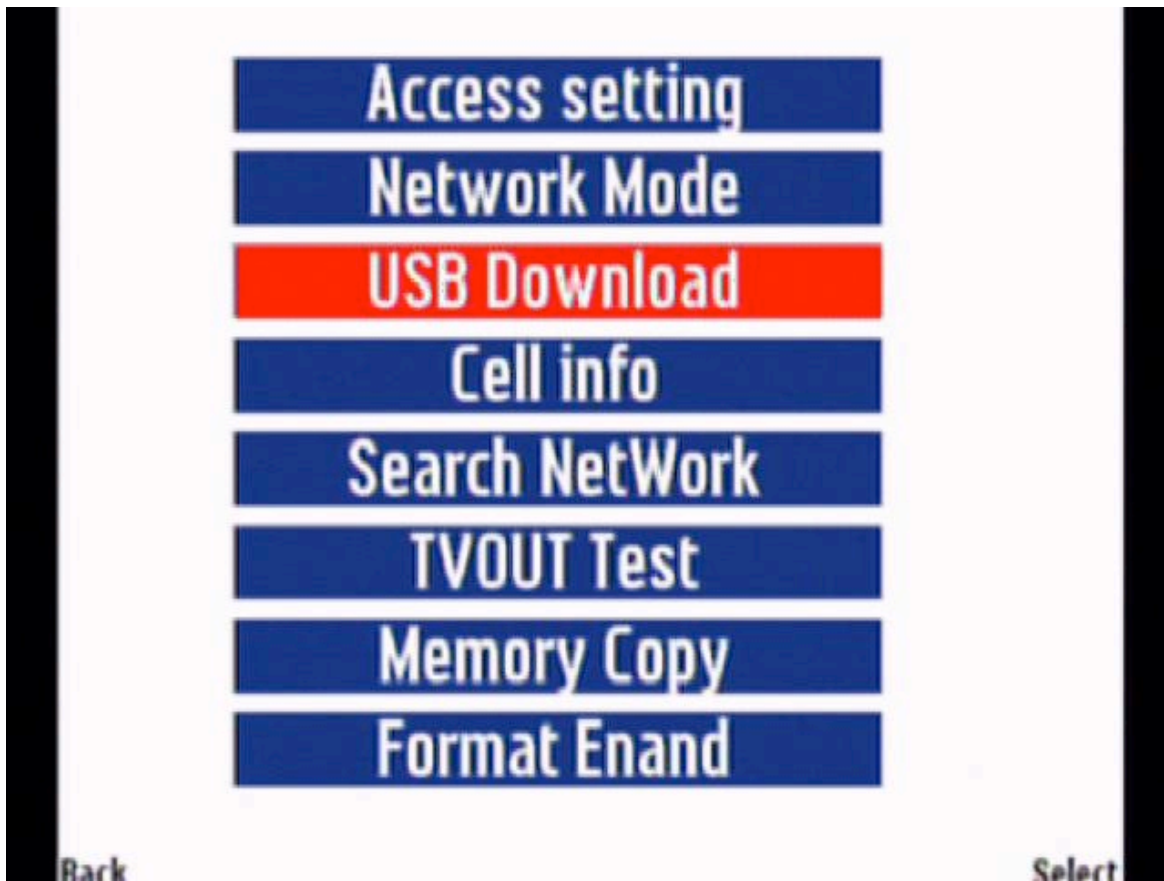
The main differences between USB Download and Trace is that in the latter case, you can use the joystick USB ports while connected in the PC. This is extremely useful while debugging or profiling games for Zeebo.

The next steps explain how to change between USB Download and Trace modes:

1. In the Main Menu, start EMAPPLET.
2. Select Field Test.



3. Select USB Download.



4. Select Download or Trace mode for the mini USB port.

Make sure to restart/reboot the console in order to get new USB port mode settings working.

11.7 Uploading games using SD card

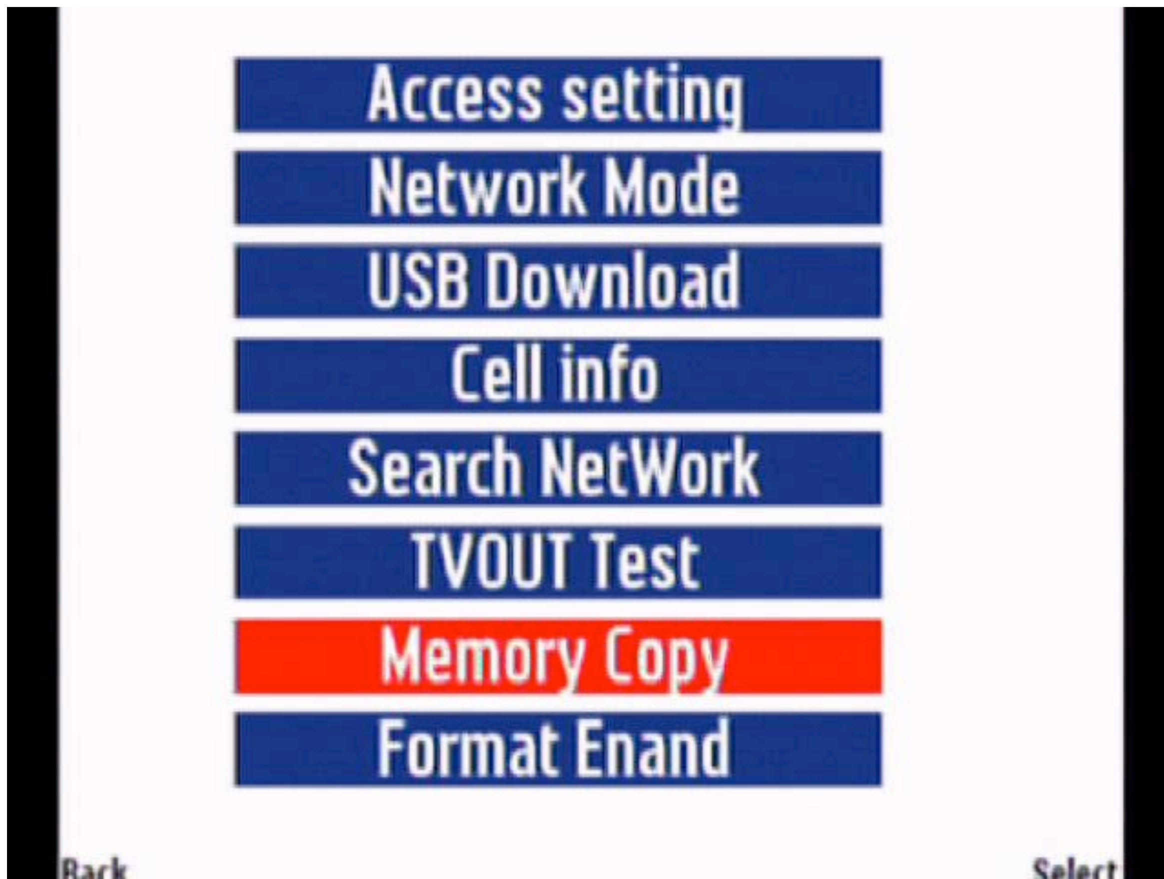
The next steps explain how to upload games using the SD memory card:

1. Create the following folder structure in your SD memory card:

   /mif ← copy .mif files here

   /mod ← root folder for games

   /mod/yourgamename ← copy your game to this folder
2. Insert the SD memory card into Zeebo.
3. In the Main Menu, start EMAPPLET.
4. Select Field Test.
5. Select Memory Copy.



The Format Enand option can be used to reset or delete all the contents previously stored in Zeebo.

11.8 Power Button Behavior

Pressing Zeebo's power button will put the device in stand by mode, rather than turning off the console. This behavior is due to scheduled updates that should occur while the console is in stand by mode. Scheduled updates include all the contents showed in Shop screen and also Zeebo User Interface.

In the games side, pressing the power button while playing a game will send a key release event AVK_END, notifying the game. Zeebo recommends that games must quit after receiving this event.

11.9 Known Issues

- Avoid using BREW function call GETTIMEMS() to retrieve or compute elapsed time for frame rate control. GETTIMEMS() takes a few more milliseconds to return the result. Instead, developers must use GETUPTIMEMS(), which is faster.

- Game's frame rate must be held down to 35 frames per second, in order to avoid flickering on the video output.

- Upon receiving EVT_APP_SUSPEND, games must release all instances associated to IHID devices (including all ISignals and ISiganlCBFactory and callbacks associated) to prevent any blocking issues. After receiving an EVT_APP_RESUME, games must create all instances associated to IHID again. These two events are received after pressing HOME button for more than 3 seconds. See section 10.1 for further details.

Zeebo Confidential and Proprietary

12 Submission process

All Zeebo games must be submitted to NSTL for TBT certification and to Zeebo Inc. for quality assurance. Please refer to Zeebo TBT NSTL test plan for more details on BREW certification for Zeebo.

Zeebo Confidential and Proprietary

Appendix A- Supported BREW API List

Category	BREW API	Zeebo Inc Requirement
Abstract Base Classes	IApplet	Yes
	IAPPLETCTL	Yes
	IAStream	Yes
	IBase	Yes
	IControl	Yes
	IModel	Yes
	IModule	Yes
	INotifier	Yes
	IParameters	Yes
	IQI	Yes
Address Book and Call History	IAddrBook	No
	IAddrRec	No
	ICallHistory	No
Application Services	IAlarmMgr	Yes
	IAppHistory	Yes
	IClipboard	Yes
	ICoreNotifier	Yes
	IDeviceNotifier	Yes
	IFIFO	Yes
	ILicense	Yes
	IMemAStream	Yes
	IRscPool	Yes
	ISignal	Yes
	IShell	Yes
	IThread	Yes
	IUnzipAStream	Yes
	ILocalStorage	Yes
	IRegistry	Yes
Backlight	IBacklight	No
Battery	IBattery	Yes
	IBatteryNotifier	Yes
Controls	IAClockCtl	Yes
	IDateCtl	Yes
	IDialog	Yes
	IImageCtl	Yes
	IMenuCtl	Yes
	IResourceCtl	Yes
	IStatic	Yes

	ITextCtl	Yes
	ITimeCtl	Yes
Database	IDatabase	Yes
	IDBMgr	Yes
	IDBRecord	Yes
Diagnostics and SIO	ILogger	Yes
	IPort	Yes
Display	IBitmap	Yes
	IBitmapDev	Yes
	IDIB	Yes
	IDisplay	Yes
	IFont	Yes
	ISprite	Yes
	IMDP	Yes
	ITransform	Yes
Download	IADSQuery	Yes
	IDownload	Yes
	IFOTA	No
File	IFile	Yes
	IFileMgr	Yes
Flip	IFlip	Yes
Image Viewers and Decoders	IForceFeed	Yes
	IImage	Yes
	IImageDecoder	Yes
	IViewer	Yes
Location-based Services	IPosdet	No
Memory Management	IHeap	Yes
	IRamCache	Yes
	IRecordStore	Yes
Multimedia	ICamera	No
	IDLS	No
	IDLSLinker	No
	IGraphics	Yes
	IHID	Yes
	IUSBHID	No
	IJoystick	No
	IMedia	Yes
	IMediaSVG	Yes
	IMediaUtil	Yes
	Multimedia_Content_File	Yes
	IRingerMgr	Yes
	ISound	Yes
	ISoundPlayer	Yes
	IEGL	Yes
	IGL	Yes
	IRender2D	Yes
	IVocoder	Optional
Network	IAddrInfo	Yes

	IAddrInfoCache	Yes
	IBCMCSDB	No
	IDNS	Yes
	IDNSConfig	Yes
	IDNSConfig2	Yes
	INetMgr	Yes
	INetwork	Yes
	INetUtils	Yes
	INetMTPDNotifier	Yes
	IMcastSession	No
	IQoSBundle	No
	IQoSFilter	No
	IQoSFlow	No
	IQoSList	No
	IQoSSession	No
	IQoSSpec	No
	ISocket	Yes
	ISockPort	Yes
	IWIFI	No
	IWIFIOpts	No
Resource Management	ITopVisibleCtl	Yes
Security	ICipher	No
	ICipher1	Yes
	ICipherFactory	Yes
	ICipherWrapper	Yes
	IHash	Yes
	IHashCTX	Yes
	IRawBlockCipher	Yes
	IRSA	Yes
	ISSL	Yes
	IX509Chain	Yes
Telephony and SMS	ICall	No
	ICallMgr	No
	ICallOrigOpts	No
	IMultipartyCall	No
	IPhoneCtl	No
	IPhoneNotifier	No
	ISMS	Yes
	ISMSBCConfig	Yes
	ISMSBCSrvOpts	Yes
	ISMSMsg	Yes
	ISMSNotifier	Yes
	ISMSStorage	Yes
	ISMSStorage2	Yes
	ISuppsTrans	Optional
	ITAPI	Yes/TBD
	ITelephone	No
Web	IGetLine	Yes

	IHtmlViewer	Yes
	IPeek	Yes
	ISource	Yes
	ISourceUtil	Yes
	IWeb	Yes
	IWebEng	Yes
	IWebOpts	Yes
	IWebResp	Yes
	IWebUtil	Yes

Appendix B – Supported OpenGL ES API List

	OpenGL ES Function	Parameter(s)	Version	Version	
			1.0	**1.1**	**Supported**
		F = float | fixed, T = int | float | fixed | ubyte | uint, clampF = clampf | clampx			Common
2.5	GetError (void)		X	X	X
		NO_ERROR	x	x	x
		INVALID_ENUM	x	x	x
		INVALID_VALUE	x	x	x
		INVALID_OPERATION	x	x	x
		STACK_OVERFLOW	x	x	x
		STACK_UNDERFLOW	x	x	x
		OUT_OF_MEMORY	x	x	x
2.7	Normal3{fx}	(T coords)	X	X	X
	MultiTexCoord4{fx}	(enum texture, T coords)	X	X	X
	Color4{fx}	(T components)	X	X	X
	Color4ub[v]	(T components)		X	
2.8	VertexPointer	(int size, enum type, sizei stride, const void *ptr)	X	X	X
		size = 2,3,4 type = BYTE	x	x	x
		size = 2,3,4 type = SHORT	x	x	x
		size = 2,3,4 type = FLOAT, FIXED	x	x	x
	NormalPointer	(enum type, sizei stride, const void *ptr)	X	X	X
		type = SHORT, BYTE	x	x	x
		type = FLOAT, FIXED	x	x	x
	ColorPointer	(int size, enum type, sizei stride, const void *ptr)	X	X	X
		size = 4 type = UNSIGNED_BYTE	x	x	x
		size = 4 type = FLOAT, FIXED	x	x	x
	TexCoordPointer	(int size, enum type, sizei stride, const void *ptr)	X	X	X
		size = 2,3,4 type = BYTE	x	x	x
		size = 2,3,4 type = SHORT	x	x	x
		size = 2,3,4 type = FLOAT, FIXED	x	x	x
	DrawArrays	(enum mode, int first, sizei count)	X	X	X
		mode = POINTS, LINES, LINE_STRIP, LINE_LOOP	x	x	x
		mode = TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN	x	x	x
	DrawElements	(enum mode, sizei count, enum type, const void *indices)	X	X	X

		mode = POINTS, LINES, LINE_STRIP, LINE_LOOP	x	x	x
		mode = TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN	x	x	x
		type = UNSIGNED_BYTE, UNSIGNED_SHORT	x	x	x
	ClientActiveTexture	(enum texture)	X	X	X
	EnableClientState	(enum cap)	X	X	X
		cap = TEXTURE_COORD_ARRAY, COLOR_ARRAY	x	x	x
		cap = NORMAL_ARRAY, VERTEX_ARRAY	x	x	x
	DisableClientState	(enum cap)	X	X	X
		cap = TEXTURE_COORD_ARRAY, COLOR_ARRAY	x	x	x
		cap = NORMAL_ARRAY, VERTEX_ARRAY	x	x	x
	WeightPointerOES	(int size, enum type, sizei stride, void *pointer)			X
	MatrixIndexPointerOES	(int size, enum type, sizei stride, void *pointer)			X
	CurrentPaletteMatrixOES	(uint index)			X
	LoadPaletteFromModelViewMatrixOES				X
		multitexture is minimum of 2 texture units		X	X
2.9	BindBuffer	(enum target, uint buffer)		X	X
	DeleteBuffers	(sizei n, uint *buffers)		X	X
	GenBuffers	(sizei n, uint *buffers)		X	X
	BufferData	(enum target, sizeiptr size, const void *data, enum usage)		X	X
	BufferSubData	(enum target, intptr offset, sizeiptr size, const void *data)		X	X
2.11	DepthRange{fx}	(clampF n, clampF f)	X	X	X
	Viewport	(int x, int y, sizei w, sizei h)	X	X	X
	MatrixMode	(enum mode)	X	X	X
		mode = MODELVIEW, PROJECTION, TEXTURE	x	x	X
	LoadMatrix{fx}	(F m[16])	X	X	X
	MultMatrix{fx}	(F m[16])	X	X	X
	LoadIdentity	(void)	X	X	X
	Rotate{fx}	(F angle, F x, F y, F z)	X	X	X
	Scale{fx}	(F x, F y, F z)	X	X	X
	Translate{fx}	(F x, F y, F z)	X	X	X
	Frustum{fx}	(F l, F r, F b, F t, F n, F f)	X	X	X
	Ortho{fx}	(F l, F r, F b, F t, F n, F f)	X	X	X
	ActiveTexture	(enum texture)	X	X	X
	PushMatrix	(void)	X	X	X
		TEXTURE and PROJECTION (2 deep)	x	x	x
		MODELVIEW (16 deep)	x	x	x
	PopMatrix	(void)	X	X	X
	Enable/Disable	(RESCALE_NORMAL)	X	X	X
	Enable/Disable	(NORMALIZE)	X	X	X
2.12	ClipPlane{fx}	(enum plane, F *equation[4])		X	

	Enable/Disable	(CLIP_PLANE{0…n-1})		X	
2.1.4	FrontFace	(enum mode)	X	X	X
	Enable/Disable	(LIGHTING)	X	X	X
	Enable/Disable	(LIGHT{0-7})	X	X	X
	Material{fx}[v]	(enum face, enum pname, T param)	X	X	X
		face = FRONT_AND_BACK	x	x	x
		pname = AMBIENT, DIFFUSE, SPECULAR, EMISSION, SHININESS	x	x	x
		pname = AMBIENT_AND_DIFFUSE	x	x	x
	Light{fx}[v]	(enum light, enum pname, T param)	X	X	X
	LightModel{fx}[v]	(enum pname, T param)	X	X	X
		pname = LIGHT_MODEL_TWO_SIDE	x	x	x
		pname = LIGHT_MODEL_AMBIENT	x	x	x
	Enable/Disable	(COLOR_MATERIAL)	X	X	X
	ShadeModel	(enum mode)	X	X	X
3.2	Enable/Disable	(MULTISAMPLE)	X	X	
3.3	PointSize[x]	(F size)	X	X	X
	PointParameter{fx}[v]	(enum pname, T param)		X	
	Enable/Disable	(POINT_SMOOTH)	X	X	X
	PointSizePointerOES	(enum type, sizei stride, const void *ptr)		X	X
		type = FLOAT, FIXED		x	x
3.4	LineWidth[x]	(F width)	X	X	x
	Enable/Disable	(LINE_SMOOTH)	X	X	x
3.5	CullFace	(enum mode)	X	X	X
	Enable/Disable	(CULL_FACE)	X	X	X
	PolygonOffset[x]	(F factor, F units)	X	X	X
	Enable/Diable	(POLYFON_OFFSET_FILL)	X	X	X
3.6	PixelStorei	(enum pname, T param)	X	X	X
		pname = PACK_ALIGNMENT, UNPACK_ALIGNMENT	x	x	x
	DrawTex{sifx}OES	(T Xs, T Ys, T Zs, T Ws, T Hs)			X
	DrawTex{sifx}OESv	(T *coords)			X
3.8	*Minimum of 2 texture units supported*			X	X
	Image Types	*UNSIGNED_BYTE*	x	x	x
		UNSIGNED_SHORT_5_6_5	x	x	x
		UNSIGND_SHORT_4_4_4_4	x	x	x
		UNSIGNED_SHORT_5_5_5_1	x	x	x
	Texture ImageFormats and Types	*RGBA, UNSIGNED_BYTE, 4*	x	x	x
	Internal/External Foramet, Type, Bytes/Pixel	*RGB, UNSIGNED_BYTE, 3*	x	x	x
		RGBA, UNSIGNED_SHORT_4_4_4_4, 2	x	x	x
		RGBA, UNSIGNED_SHORT_5_5_5_1, 2	x	x	x
		RGB, UNSIGNED_SHORT_5_6_5, 2	x	x	x
		LUMINANCE_ALPHA, UNSIGNED_BYTE, 2	x	x	x

		LUMINANCE UNSIGNED_BYTE, 1	x	x	x
		ALPHA, UNSIGNED_BYTE, 1	x	x	x
	Image Copy Conversions	*ALPHA -> APLHA*	x	x	x
	Color Buffer -> Texture IinternalFormat	*LUMINANCE -> LUMINANCE*	x	x	x
		LUMINANCE_ALPHA -> ALPHA, LUMINANCE, LUMINANCE_ALPHA	x	x	x
		RGB -> LUMINANCE, RGB	x	x	x
		RGBA -> ALPHA, LUMINANCE, LUMINANCE_ALPHA, RGB, RGBA	x	x	x
3.8.3	*TEXTURE_WRAP_S, TEXTURE_WRAP_T*	*REPEAT*	x	x	x
		CLAMP_TO_EDGE	x	x	x
	TEXTURE_MIN_FILTER	*NEAREST*	x	x	x
		LINEAR	x	x	x
		NEAREST_MIPMAP_NEAREST	x	x	x
		NEAREST_MIPMAP_LINEAR	x	x	x
		LINEAR_MIPMAP_NEAREST	x	x	x
		LINEAR_MIPMAP_LINEAR	x	x	x
	TEXTURE_MAG_FILTE R	*NEAREST*	x	x	x
		LINEAR	x	x	x
3.8.5	TexImage2D	(enum target, int level, int internalFormant, sizei width, sizei height, int border, enum format, enum type, const void *pixels)	X	X	X
		target = TEXTURE_2D, border = 0	x	x	x
	TexSubImage2D	(enum target, int level, int xoffset, int yoffset, sizei width, sizei height, enum format, enum type, const void *pixels)	X	X	X
	CopyTexImage2D	(enum target, int level, enum internalformat, int x, int y, sizei width, sizei height, int border)	X	X	X
		border = 0	X	X	x
	CopyTexSubImage2D	(enum target, int level, int xoffset, int x, int y, sizei width, sizei height)	X	X	X
	CompressedTexImage2 D	(enum target, int level, enum internalFormat, sizei width, sizei height, int border, sizei imageSize, const void *data)	X	X	X
	Paletted Compressed Texture formats	*target = TEXTURE2D, border = 0*	x	x	x
		internalFormat = PALETTE4_RGB8_OES	x	x	x
		internalFormat = PALETTE4_RGBA8_OES	x	x	x
		internalFormat = PALETTE4_R5_G6_B5_OES	x	x	x
		internalFormat = PALETTE4_RGBA4_OES	x	x	x
		internalFormat = PALETTE4_RGB5_A1_OES	x	x	x
		internalFormat = PALETTE8_RGB8_OES	x	x	x
		internalFormat = PALETTE8_RGBA8_OES	x	x	x
		internalFormat = PALETTE8_R5_G6_B5_OES	x	x	x

		internalFormat = PALETTE8_RGBA4_OES	x	x	x
		internalFormat = PALETTE8_RGB5_A1_OES	x	x	x
	CompressedTexSubImage2D	(enum target, int level, int xoffset, sizei width, enum format, sizei imageSize, const void *data)	X	X	x
	TexParameter{fx}[v]	(enum target, enum pname, T param)	X		X
		internalFormat = GL_COMPRESSED_RGB_ATI_TC			x
		internalFormat = GL_COMPRESSED_RGBA_ATI_TC			x
		target = TEXTURE_2D	x		x
		target = TEXTURE_MIN_FILTER, TEXTURE_MAG_FILTER	x		x
		target = TEXTURE_WRAP_S, TEXTURE_WRAP_T	x		x
	TexParameter{ifx}[v]	(enum target, enum pname, T params)		X	X
		pname = TEXTURE_CROP_RECT_OES			x
		target = TEXTURE2D		x	x
		pname = TEXTURE_MIN_FILTER, TEXTURE_MAG_FILTER		x	x
		pname = TEXTURE_WRAP_S, TEXTURE_WRAP_T		x	x
		pname = GENERATE_MIPMAP		x	
		param = MIRRORED_REPEAT			x
	BindTexture	(enum target, uint texture)	X	X	X
		target = TEXTURE_2D	x	x	x
	DeleteTextures	(sizei n, uint *textures)	X	X	X
	GenTextures	(sizei n, uint *textures)	X	X	X
	Enable/Disable	(TEXTURE_2D)	X	X	X
	TexEnv{ifx}[v]	(enum target, enum pname, T param)	X	X	X
		pname = TEXTURE_ENV_COLOR	x	x	x
		pname = TEXTURE_ENV_MODE	x	x	x
		param = MODULATE, REPLACE, DECAL	x	x	x
		param = BLEND, ADD	x	x	x
		param = COMBINE		x	x
		pname = COMBINE_RGB, COMBINE_ALPHA		x	x
		pname = SRC{012}_RGB, SRC{012}_ALPHA		x	x
		param = DOT3_RGB, DOT3_RGBA			x
		pname = OPERAND{012}_RGB, OPERAND{012}_ALPHA		x	x
		param = TEXTURE			x
		pname = RGB_SCALE, ALPHA_SCALE		x	x
		target = POINT_SPRITE_OES		x	x
		pname = COORD_REPLACE_OES		x	x
		param = {TRUE | FALSE }		x	x
3.9	Fog{fx}[v]	(enum pname, T param)	X	X	X

		pname = FOG_MODE, FOG_DENSITY, FOG_START, FOG_END, FOG_COLOR	x	x	x
	Enable/Disable	(FOG)	X	X	x
4.1	Enable/Disable	(SCISSOR_TEST)	X	X	x
	Scissor	(int x, int y, sizei width, sizei height)	X	X	X
	Enable/Disable	(SAMPLE_COVERAGE)	X	X	
	Enable/Disable	(SAMPLE_ALPHA_TO_COVERAGE)	X	X	
	Enable/Disable	(SAMPLE_ALPHA_TO_ONE)	X	X	
	SampleCoverage[x]	(clampF value, boolean invert)	X	X	
	Enable/Disable	(ALPHA_TEST)	X	X	X
	AlphaFunc[x]	(enum func, clampF ref)	X	X	X
	Enable/Disable	(STENCIL_TEST)	X	X	X
	StencilFunc	(enum func, int ref, uint mask)	X	X	X
	StencilMask	(unint mask)	X	X	X
	StencilOp	(enum fail, enum zfail, enum zpass)	x	x	x
		fail, zfail, zpass = KEEP	x	x	x
		fail, zfail, zpass = ZERO	x	x	x
		fail, zfail, zpass = REPLACE	x	x	x
		fail, zfail, zpass = INCR	x	x	x
		fail, zfail, zpass = DECR	x	x	x
		fail, zfail, zpass = INVERT	x	x	x
	Enable/Disable	(DEPTH_TEST)	X	X	X
	DepthFunc	(enum func)	X	X	X
	DepthMask	(boolean flag)	X	X	X
	Enable/Disable	(BLEND)	X	X	X
	BlendFunc	(enum sfactor, enum dfactor)	X	X	X
	Enable/Disable	(DITHER)	X	X	X
	Enable/Disable	(COLOR_LOGIC_OP)	X	X	X
	LogicOp	(enum opcode)	X	X	X
4.2	ColorMask	(boolean red, boolean green, boolean blue, boolean alpha)	X	X	X
	Clear	S(bitfield mask)	X	X	X
	ClearColor[x]	(clampF red, clampF green, clampF blue, clampF alpha)	X	X	X
	ColorDepth{fx}	(clampF depth)	X	X	X
	ClearStencil	(int s)	X	X	X
4.3	ReadPixels	(intx, int y, sizei width, sizei height, enum format, enum type, void *pixels)	X	X	X
5.5	Flush	(void)	X	X	X
	Finish	(void)	X	X	X
5.6	Hint	(enum target, enum mode)	X	X	X
		target = PERSPECTIVE_CORRECTION_HINT	x	x	x
		target = POINT_SMOOTH_HINT	x	x	x
		target = LINE_SMOOTH_HINT	x	x	x
		target = FOG_HINT	x	x	x
		target = GENERATE_MIPMAP_HINT		x	

Zeebo Confidential and Proprietary

6.1	GetBooleanv	(enum pname, boolean *params)		X	
	GetIntegerv	(enum pname, int *params)	X	X	X
	Get{Float|Fixed}v	(enum pname, T *params)		X	
	IsEnabled	(enum cap)	X	X	
	GetClipPlane{fx}	(enum plane, T equation[4])		X	
	GetLight{fx}v	(enum light, enum pname, T *params)		X	
	GetMaterial{fx}v	(enum face, enume pname, T *params)		X	X
	GetTexEnv{ifx}v	(enum target, enum pname, T *params)		X	
	GetTexParameter{ifx}v	(enum target, enum pname, T *params)		X	X
	GetBufferParameteriv	(enum target, enum pname, boolean, *params)		X	X
	IsTexture	(unit texture)		X	
	GetPointerv	(enum pname, void **params)		X	X
	GetString	(enume name)	X	X	X
	IsBuffer	(uint buffer)		X	X
6.2	*Queryable State*				
	GetIntegerv Get{Float|Fixed}v	*CURRENT_COLOR*		X	
	Get{Float|Fixed}v	*CURRENT_TEXTURE_COORDS*		X	
	Get{Float|Fixed}v	*CURRENT_NORMAL*		X	
6.6	GetIntegerv	*CLIENT_ACTIVE_TEXTURE*		X	
	IsEnabled	*VERTEX_ARRAY*		X	
	GetIntegerv	*VERTEX_ARRAY_SIZE*		X	
	GetIntegerv	*VERTEX_ARRAY_STRIDE*		X	
	GetIntegerv	*VERTEX_ARRAY_TYPE*		X	
	GetPointerv	*VERTEX_ARRAY_POINTER*		X	X
	IsEnabled	*NORMAL_ARRAY*		X	
	GetIntegerv	*NORMAL_ARRAY_STRIDE*		X	
	GetIntegerv	*NORMAL_ARRAY_TYPE*		X	
	GetPointerv	*NORMAL_ARRAY_POINTER*		X	X
	IsEnabled	*COLOR_ARRAY*		X	
	GetIntegerv	*COLOR_ARRAY_SIZE*		X	
	GetIntegerv	*COLOR_ARRAY_STRIDE*		X	
	GetIntegerv	*COLOR_ARRAY_TYPE*		X	
	GetPointerv	*COLOR_ARRAY_POINTER*		X	X
	IsEnabled	*TEXTURE_COORD_ARRAY*		X	
	GetIntegerv	*TEXTURE_COORD_ARRAY_SIZE*		X	
	GetIntegerv	*TEXTURE_COORD_ARRAY_STRIDE*		X	
	GetIntegerv	*TEXTURE_COORD_ARRAY_TYPE*		X	
	GetPointerv	*TEXTURE_COORD_ARRAY_POINTER*		X	X
	GetIntegerv	*ARRAY_BUFFER_BINDING*		X	
	GetIntegerv	*VERTEX_ARRAY_BUFFER_BINDING*		X	
	GetIntegerv	*NORMAL_ARRAY_BUFFER_BINDING*		X	
	GetIntegerv	*COLOR_ARRAY_BUFFER_BINDING*		X	
	GetIntegerv	*TEXTURE_COORD_ARRAY_BUF*		X	

		FER_BINDING			
	GetIntegerv	*ELEMENT_ARRAY_BUFFER_BINDING*		X	
6.7	GetBufferParameteriv	*BUFFER_SIZE*		X	X
	GetBufferParameteriv	*BUFFER_USAGE*		X	X
	GetBufferParameteriv	*BUFFER_ACCESS*		X	X
6.8	Get{Float|Fixed}v	*MODELVIEW_MATRIx*		X	X
	Get{Float|Fixed}v	*PROJECTION_MATRIX*		X	X
	Get{Float|Fixed}v	*TEXTURE_MATRIX*		X	X
	GetIntegerv	*MODELVIEW_MATRIX_FLOAT_AS_INT_BITS_OES*		X	
	GetIntegerv	*PROJECTION_MATRIX_FLOAT_AS_INT_BITS_OES*		X	
	GetIntegerv	*TEXTURE_MATRIX_FLOAT_AS_INT_BITS_OES*		X	
	GetIntegerv	*VIEWPORT*		X	
	Get{Float|Fixed}v	*DEPTH_RANGE*		X	
	GetIntegerv	*MODELVIEW_STACK_DEPTH*		X	X
	GetIntegerv	*PROJECTION_STACK_DEPTH*		X	X
	GetIntegerv	*TEXTURE_STACK_DEPTH*		X	X
	GetIntegerv	*MATRIX_MODE*		X	
	IsEnabled	*NORMALIZE*		X	
	IsEnabled	*RESCALE_NORMAL*		X	
	GetClipPlane{fx}	*CLIP_PLANE{0-5}*		X	
	IsEnabled	*CLIP_PLANE{0-5}*		X	
6.9	Get{Float|Fixed}v	*FOG_COLOR*		X	
	Get{Float|Fixed}v	*FOG_DENSITY*		X	
	Get{Float|Fixed}v	*FOG_START*		X	
	Get{Float|Fixed}v	*FOG_END*		X	
	GetIntegerv	*FOG_MODE*		X	
	IsEnabled	*FOG*		X	
	GetIntegerv	*SHADE_MODEL*		X	
6.10	IsEnabled	*LIGHTING*		X	
	IsEnabled	*COLOR_MATERIAL*		X	
	GetMaterial{fx}v	*AMBIENT (material)*		X	X
	GetMaterial{fx}v	*DIFFUSE (material)*		X	X
	GetMaterial{fx}v	*SPECULAR (material)*		X	X
	GetMaterial{fx}v	*EMISSION (material)*		X	X
	GetMaterial{fx}v	*SHININESS (material)*		X	X
	Get{Float|Fixed}v	*LIGHT_MODEL_AMBIENT*		X	
	GetBooleanv	*LIGHT_MODEL_TWO_SIDE*		X	
	GetLight{fx}v	*AMBIENT* (light$_i$)		X	
	GetLight{fx}v	*DIFFUSE* (light$_i$)		X	
	GetLight{fx}v	*SPECULAR* (light$_i$)		X	
	GetLight{fx}v	*POSITION* (light$_i$)		X	
	GetLight{fx}v	*CONSTANT_ATTENUATION*		X	
	GetLight{fx}v	*LINEAR_ATTENUATION*		X	
	GetLight{fx}v	*QUADRATIC_ATTENUATION*		X	
	GetLight{fx}v	*SPOT_DIRECTION*		X	
	GetLight{fx}v	*SPOT_EXPONENT*		X	

	GetLight{fx}v	*SPOT_CUTOFF*		X	
	IsEnabled	*LIGHT{0-7}*		X	
6.11	Get{Float|Fixed}v	*POINT_SIZE*		X	
	IsEnabled	*MOINT_SMOOTH*		X	
	Get{Float|Fixed}v	*POINT_SIZE_MIN*		X	
	Get{Float|Fixed}v	*POINT_SIZE_MAX*		X	
	Get{Float|Fixed}v	*POINT_FADE_THRESHOLD_SIZE*		X	
	Get{Float|Fixed}v	*POINT_DISTANCE_ATTENUATION*		X	
	Get{Float|Fixed}v	*LINE_WIDTH*		X	
	IsEnabled	*LINE_SMOOTH*		X	
	IsEnabled	*CULL_FACE*		X	
	GetIntegerv	*CULL_FACE_MODE*		X	
	GetIntegerv	*FRONT_FACE*		X	
	Get{Float|Fixed}v	*POLYGON_OFFSET_FACTOR*		X	
	Get{Float|Fixed}v	*POLYGON_OFFSET_UNITS*		X	
	IsEnabled	*POLYGON_OFFSET_FILL*		X	
6.12	IsEnabled	*MULTISAMPLE*		X	
	IsEnabled	*SAMPLE_ALPHA_TO_COVERAGE*		X	
	IsEnabled	*SAMPLE_ALPHA_TO_ONE*		X	
	IsEnabled	*SAMPLE_COVERAGE*		X	
	Get{Float|Fixed}v	*SAMPLE_COVERAGE_VALUE*		X	
	GetBooleanv	*SAMPLE_COVERAGE_INVERT*		X	
6.13	IsEnabled	*TEXTURE_2D*		X	
	GetIntegerv	*TEXTURE_BINGDING_2D*		X	
	GetTexParameteriv	*TEXTURE_MIN_FILTER*		X	X
	GetTexParameteriv	*TEXTURE_MAG_FILTER*		X	X
	GetTexParameteriv	*TEXTURE_WRAP_S*		X	X
	GetTexParameteriv	*TEXTURE_WRAP_T*		X	X
	GetTexParameteriv	*GENERATE_MIPMAP*		X	
6.14	GetIntegerv	*ACTIVE_TEXTURE*		X	
	GetTexEnviv	*TEXTURE_ENV_MODE*		X	
	GetTexEnv{fx}v	*TEXTURE_ENV_COLOR*		X	
	GetTexEnviv	*COMBINE_RGB*		X	
	GetTexEnviv	*COMBINE_ALPHA*		X	
	GetTexEnviv	*SRC{012}_RGB*		X	
	GetTexEnviv	*SRC{012}_ALPHA*		X	
	GetTexEnviv	*OPERAND{012}_RGB*		X	
	GetTexEnviv	*OPERAND{012}_ALPHA*		X	
	GetTexEnviv	*RGB_SCALE*		X	
	GetTexEnviv	*ALPHA_SCALE*		X	
6.15	GetBooleanv	*COLOR_WRITEMASK*		X	
	GetBooleanv	*DEPTH_WRITEMASK*		X	
	GetIntegerv	*STENCIL_WRITEMASK*		X	
	Get{Float|Fixed}v	*COLOR_CLEAR_VALUE*		X	
	GetIntegerv	*DEPTH_CLEAR_VALUE*		X	
	GetIntegerv	*STENCIL_CLEAR_VALUE*		X	

Zeebo Confidential and Proprietary

6.16	IsEnabled	*SCISSOR_TEST*		X	
	GetIntegerv	*SCISSOR_BOX*		X	
	IsEnabled	*ALPHA_TEST*		X	
	GetIntegerv	*ALPHA_TEST_FUNC*		X	
	GetIntegerv	*ALPHA_TEST_REF*		X	
	IsEnabled	*STENCIL_TEST*		X	
	GetIntegerv	*STENCIL_FUNC*		X	
	GetIntegerv	*STENCIL_VALUE_MASK*		X	
	GetIntegerv	*STENCIL_REF*		X	
	GetIntegerv	*STENCIL_FAIL*		X	
	GetIntegerv	*STENCIL_PASS_DEPTH_FAIL*		X	
	GetIntegerv	*STENCIL_PASS_DEPTH_PASS*		X	
	IsEnabled	*DEPTH_TEST*		X	
	GetIntegerv	*DEPTH_FUNC*		X	
	IsEnabled	*BLEND*		X	
	GetIntegerv	*BLEND_SRC*		X	
	GetIntegerv	*BLEND_DST*		X	
	IsEnabled	*DITHER*		X	
	IsEnabled	*COLOR_LOGIC_OP*		X	
	GetIntegerv	*LOGIC_OP_MODE*		X	
6.17	GetIntegerv	*UNPACK_ALIGNMENT*		X	
	GetIntegerv	*PACK_ALIGNMENT*		X	
	GetIntegerv	*PERSPECTIVE_CORRECTION_HINT*	X	X	
	GetIntegerv	*POINT_SMOOTH_HINT*	X	X	
	GetIntegerv	*LINE_SMOOTH_HINT*	X	X	
		TEXTURE_COMPRESSION_HINT	X		
	GetIntegerv	*FOG_HINT*	X	X	
	GetIntegerv	*GENERATE_MIPMAP_HINT*		X	
6.24	GetIntegerv	*MAX_CLIP_PLANES*		X	
	GetIntegerv	*MAX_MODELVIEW_STACK_DEPTH*	X	X	X
	GetIntegerv	*MAX_PROJECTION_STACK_DEPTH*	X	X	X
	GetIntegerv	*MAX_TEXTURE_STACK_DEPTH*	X	X	X
	GetIntegerv	*SUPIXEL_BLITS*	X	X	X
	GetIntegerv	*MAX_TEXTURE_SIZE*	X	X	X
	GetIntegerv	*MAX_VIEWPORT_DIMS*	X	X	X
6.25	Get{Float|Fixed}v	*ALIASED_POINT_SIZE_RANGE*	X	X	
	Get{Float|Fixed}v	*SMOOTH_POINT_SIZE_RANGE*	X	X	
	Get{Float|Fixed}v	*ALIASED_LINE_WIDTH_RANGE*	X	X	
	Get{Float|Fixed}v	*SMOOTH_LINE_WIDTH_RANGE*	X	X	
6.26	GetIntegerv	*MAX_ELEMENTS_INDICES*		X	X
	GetIntegerv	*MAX_ELEMENTS_VERTICES*		X	X
	GetIntegerv	*MAX_TEXTURE_UNITS*	X	X	X
	GetIntegerv	*SAMPLE_BUFFERS*	X	X	
	GetIntegerv	*SAMPLES*	X	X	
	GetIntegerv	*COMPRESSED_TEXTURE_FORMATS*	X	X	X
	GetIntegerv	*NUM_COMPRESSED_TEXTURE_FORMATS*	X	X	X

Zeebo Confidential and Proprietary

6.27	GetIntegerv	*RED_BITS*	X	X	X
	GetIntegerv	*GREEN_BITS*	X	X	X
	GetIntegerv	*BLUE_BITS*	X	X	X
	GetIntegerv	*ALPHA_BITS*	X	X	X
	GetIntegerv	*DEPTH_BITS*	X	X	X
	GetIntegerv	*STENCIL_BITS*	X	X	X
6.28	GetError	*Current Error Code(s)*	X	X	X
6.29	GetIntegerv	*IMPLEMENTATION_COLOR_READ_TYPE_OES*	X	X	X
	GetIntegerv	*IMPLEMENTATION_COLOR_READ_FORMAT_OES*	X	X	X
	IsEnabled	*MATRIX_PALETTE_OES*			
	GetIntegerv	*MAX_PALETTE_MATRICES_OES*			X
	GetIntegerv	*MAX_VERTEX_UNITS_OES*			X
	IsEnabled	*MATRIX_INDEX_ARRAY_OES*			
	GetIntegerv	*MATRIX_INDEX_ARRAY_SIZE_OES*			X
	GetIntegerv	*MATRIX_INDEX_ARRAY_TYPE_OES*			X
	GetIntegerv	*MATRIX_INDEX_ARRAY_STRIDE_OES*			X
	GetPointerv	*MATRIX_INDEX_ARRAY_POINTER_OES*			X
	GetIntegerv	*MATRIX_INDEX_ARRAY_BUFFER_BINDING_OES*			X
	IsEnabled	*WEIGHT_ARRAY_OES*			
	GetIntegerv	*WEIGHT_ARRAY_SIZE_OES*			X
	GetIntegerv	*WEIGHT_ARRAY_TYPE_OES*			X
	GetIntegerv	*WEIGHT_ARRAY_STRIDE_OES*			X
	GetIntegerv	*WEIGHT_ARRAY_POINTER_OES*			X
	GetIntegerv	*WEIGHT_ARRAY_BUFFER_BINDING_OES*			X
	GetIntegerv	*CURRENT_PALETTE_MATRIX_OES*			
	IsEnabled	*POINT_SPRITE_OES*		X	
	GetTexEnviv	*COORD_REPLACE_OES*		X	
	IsEnabled	*POINT_SIZE_ARRAY_OES*		X	
	GetIntegerv	*POINT_SIZE_ARRAY_TYPE_OES*		X	
	GetIntegerv	*POINT_SIZE_ARRAY_STRIDE_OES*		X	
	GetPointerv	*POINT_SIZE_ARRAY_POINTER_OES*		X	
	GetIntegerv	*POINT_SIZE_ARRAY_BUFFER_BINDING_OES*		X	
	GetTexParameteriv	*TEXTURE_CROP_RECT_OES*			X
0.0	QueryMatrixxOES	(GLfixed mantissa[16], GLint exponent[16])	X		
	BlendEquationEXT, BlendEquationSeparateEXT	*(enum, enum)*			X
		GL_FUNC_ADD_EXT			x
		GL_FUNC_SUBTRACT_EXT			x
		GL_FUNC_REVERSE_SUBTRACT_EXT			x
		GL_MIN_EXT			x

		GL_MAX_EXT			x
	BlendFuncSeparateEXT				X
		GL_ZERO			x
		GL_ONE			x
		GL_SRC_COLOR			x
		GL_ONE_MINUS_SRC_COLOR			x
		GL_DST_COLOR			x
		GL_ONE_MINUS_DST_COLOR			x
		GL_SRC_ALPHA			x
		GL_ONE_MINUS_SRC_ALPHA			x
		GL_DST_ALPHA			x
		GL_ONE_MINUS_DST_ALPHA			x
		GL_SRC_ALPHA_SATURATE			x
	DrawVertexBufferObject ATI				X
	MeshListATI	*int numlists, enum mode[], Tcount[], enum type[], const GLvoid *indices[]*			X

Appendix C – List of Acronyms

BLT – Bit Lookup Table

CMX - Compact Media Extensions

ZEEBO HDK – Form Factor Accurate

GPU - Graphics Processing Unit

HID - Human Interface Device

MSM - Mobile Station Modem

Zeebo Confidential and Proprietary