V M L A B S

MML2D
2D Graphics Library
For

Revision 2.63
19-Dec-01

VM Labs, Inc.
520 San Antonio Rd
Mountain View, CA 94040

Tel: (650) 917-8050
Fax: (650) 917-8052

Copyright © 1999-2001 VM Labs, Inc. All rights reserved.

NUON, NUON Media Architecture, the NUON logo and the VM Labs logo are trademarks of VM Labs, Inc.

Proprietary and Confidential to VM Labs, Inc.

The information contained in this document is confidential and proprietary to VM Labs, Inc. It may not be distributed or copied in any form whatsoever without the prior written permission of VM Labs.

This is a preliminary specification. VM Labs reserves the right to make changes to any information in this document.

Table of Contents

1. INTRODUCTION 1-1

2. OVERVIEW OF MML2D 2-1

2.1 API Layers 2-1
2.1.1 Low Level Direct Mode 2-1
2.1.2 High Level List Mode 2-2
2.1.3 MPE Rendering Functions 2-2
2.2 The Rendering Process 2-2

3. BASIC DATA TYPES 3-1

3.1 System Information 3-1
3.1.1 mmlSysResources 3-1
3.2 Style & Color Values 3-1
3.2.1 mmlColor 3-1
3.2.2 m2dLineStyle 3-1
3.2.3 m2dEllipseStyle 3-3
3.2.4 mmlPoint 3-4
3.2.5 m2dPoint 3-5
3.2.6 m2dRect 3-5
3.2.7 m2dBox 3-5
3.2.8 mmlBox 3-5
3.2.9 mmlLayoutMetrics 3-5
3.2.10 Miscellaneous Enumerated Types 3-6
3.3 Graphics Context 3-6
3.3.1 mmlGC 3-6
3.4 Bitmap Definitions 3-11
3.4.1 mmlPixmap 3-11
3.4.2 mmlAppPixmap 3-12
3.4.3 mmlDisplayPixmaps 3-12
3.5 Command Sequences 3-12
3.5.1 mmlSequence 3-12

4. GENERAL MEDIA LIBRARY FUNCTIONS 4-1

4.1 Pixmap Initializers and Attribute Setting 4-1
4.1.1 mmlInitAppPixmaps 4-1
4.1.2 mmlInitDisplayPixmaps 4-2
4.1.3 mmlReleasePixmaps 4-3
4.1.4 mmlSetPixmapClut 4-3
4.2 Other Initialization 4-3
4.2.1 mmlPowerUpGraphics 4-3
4.2.2 mmlInitGC 4-3

4.2.3 mmlSimpleVideoSetup 4-3
4.2.4 mmlInitFontContext 4-4
4.2.5 m2dSetPoint 4-4
4.2.6 m2dSetRect 4-4

5. COLOR ALLOCATION AND MANIPULATION 5-1

5.1.1 mmlColorFromRGB 5-1
5.1.2 mmlColorFromYCC 5-2
5.1.3 mmlGetYCCComponents 5-2
5.1.4 mmlColorFromYCCf 5-2
5.1.5 mmlGetYCCFloatComponents 5-2
5.1.6 mmlGetYCCComponents 5-3
5.1.7 mmlColorFromRGBf 5-3
5.1.8 mmlGetRGBFloatComponents 5-3
5.1.9 mmlGetRGBComponents 5-3
5.1.10 mmlSafeColor 5-3
5.1.11 mmlSafeColorLimits 5-4
5.1.12 mmlCustomSafeColorLimits 5-4

6. COPY FUNCTIONS 6-1

6.1.1 m2dReadPixels 6-1
6.1.2 m2dWritePixels 6-1
6.1.3 m2dCopyRect 6-2
6.1.4 m2dCopyClutRect 6-2
6.1.5 m2dCopyRectScaled 6-2
6.1.6 m2dCopyRectFast 6-3
6.1.7 m2dCopyRectDis 6-5

7. FILL FUNCTIONS 7-1

7.1.1 m2dDrawPoint 7-1
7.1.2 m2dSmallFill 7-1
7.1.3 m2dFillColor 7-1
7.1.4 m2dFillClut 7-2

8. LINE DRAW FUNCTIONS 8-1

8.1.1 m2dInitLineStyle 8-1
8.1.2 m2dDrawLine 8-1
8.1.3 m2dDrawStyledLine 8-1
8.1.4 m2dDrawPolyLine 8-1

9. ELLIPSE DRAW FUNCTIONS 9-1

9.1.1 m2dInitEllipseStyle 9-1
9.1.2 m2dDrawEllipse 9-1
9.1.3 m2dDrawStyledEllipse 9-1
9.1.4 m2dDrawQuadArc 9-2

9.1.5 m2dDrawStyledQuadArc 9-2

10. BOX FUNCTIONS 10-1

10.1.1 m2dInitBox 10-1
10.1.2 m2dDrawBox 10-1
10.1.3 m2dEraseBox 10-1
10.1.4 m2dRedrawBox 10-1
10.1.5 m2dReleaseBox 10-2

11. ARROW FUNCTIONS 11-1

11.1.1 m2dInitArrow 11-1
11.1.2 m2dSetArrowPixel 11-1
11.1.3 m2dShowArrow 11-1
11.1.4 m2dHideArrow 11-1
11.1.5 m2dRedrawArrow 11-2
11.1.6 m2dMoveArrow 11-2
11.1.7 m2dDeleteArrow 11-2

12. TEXT AND FONTS 12-1

12.1.1 mmlGetRegisteredFonts 12-1
12.1.2 mmlAddFont 12-2
12.1.3 mmlRemoveFont 12-2
12.1.4 mmlGetFontName 12-2
12.1.5 mmlSetTextProperties 12-2
12.1.6 mmlInitScaledTextStyle 12-3
12.1.7 mmlInitTextStyle (deprecated) 12-4
12.1.8 mmlSetTextStyle 12-4
12.1.9 mmlSetTextModel 12-4
12.1.10 mmlSimpleDrawText 12-5
12.1.11 mmlSimpleDrawBaseline 12-5
12.1.12 mmlGetTextBox 12-5
12.1.13 mmlGetStyleLayoutMetrics 12-6
12.1.14 mmlGetLayoutMetrics 12-6
12.1.15 mmlCharKindQ 12-6

13. SEQUENCE FUNCTIONS 13-1

13.1.1 mmlOpenSeq 13-1
13.1.2 mmlCloseSeq 13-1
13.1.3 mmlExecuteSeq 13-1
13.1.4 mmlReopenSeq 13-1
13.1.5 mmlReleaseSeq 13-2

This page intentionally left blank.

1. Introduction

The Merlin Media Library (MML) is a rich set of functions and objects that provide programmers with tools for building exciting multimedia applications for NUON.[1]

The goal of the MML is to make it possible to seamlessly integrate text, graphics (both 2D and 3D), video (including MPEG), and audio in applications.

The highest-level MML functions are C APIs for 2D and 3D graphics, MPEG and other video playback, and for audio playback. These APIs are also accessible from other high level languages (such as C++ and Pascal) that run on Merlin Media Architecture compliant devices.

There is a single direct mode API for audio functions, incorporating both MIDI and digital audio.

The movie API is a high level interface for playing back movies, including both video and audio.

There are two C APIs for graphics: the **direct mode** API, in which drawing commands for 2D and 3D primitives are issued directly, and the **list mode** API in which a single C function call draws a list of primitives that can include 2D objects, 3D objects, and movies. These primitives can be combined in a hierarchy, so that for example an MPEG-1 video could be used as a texture on the side of a rotating 3D cube.

Also provided in the MML are low-level assembly language functions that directly access the Merlin Processing Elements (MPEs). Programmers will not usually need to use these low-level functions, but they are available for developers with particular performance needs who wish to customize their applications or to achieve higher levels of performance than are possible with the high level APIs.

The MML may be used for graphics and audio output by different kinds of applications.

Figure 1-1 shows the three kinds of applications: host applications without any NUON specific code, mixed NUON and host (host application with MML libraries), and native NUON (MMA compliant).

---

[1] The original “nickname” for NUON used internally at VM Labs was “MERLIN”. This is still reflected in the names of things like development tools, libraries, and related documentation.

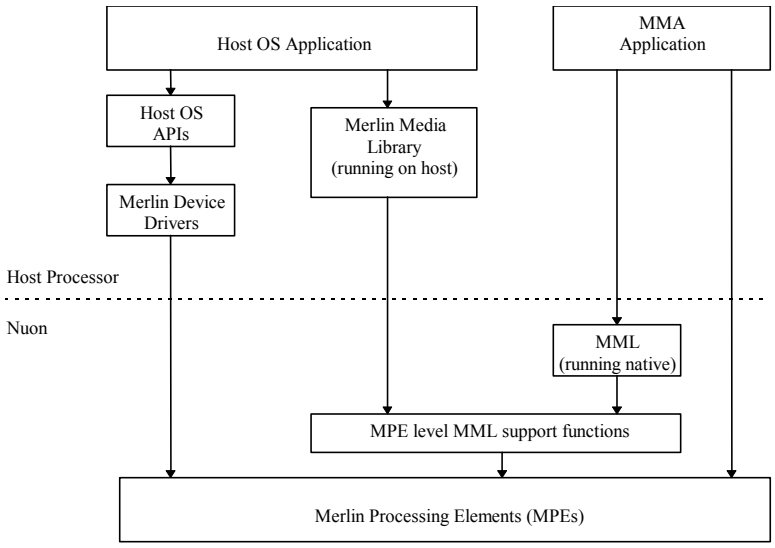

Figure 1-1, MML Application Types

Applications running on a system with a host processor and operating system (such as the Motorola Blackbird platform with the Microware DAVID operating system) can be linked with MML libraries in order to achieve higher performance or to take advantage of NUON specific features, at the cost of portability. Such applications may also use that operating system's input facilities for interacting with the user and may mix MML calls and OS output calls using a NUON device driver.

Applications that run entirely on the NUON chip will typically use MML libraries running native on NUON.

Applications of both types may make NUON BIOS calls to read user input such as joystick events (see the NUON BIOS documentation).

2. Overview of MML2D

2.1 API Layers

There are three layers in the MML2D graphics API: the MPE rendering functions, low level (direct mode) functions with a C language interface, and high level (list mode) functions with a C language interface.

Each layer is implemented in terms of layers underneath it: the high level functions call the direct mode functions, and both call MPE rendering functions directly (or indirectly).

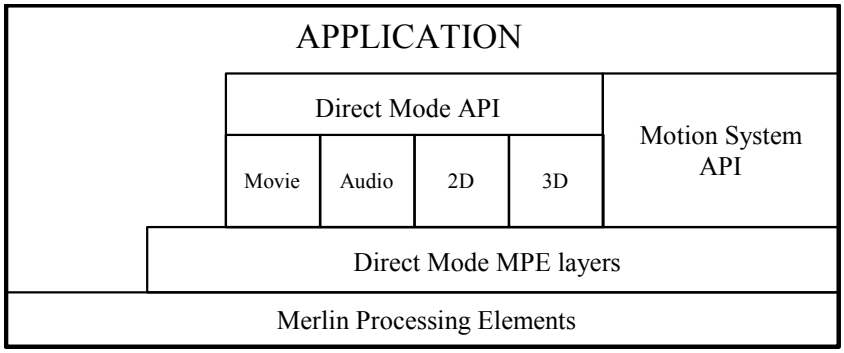

Figure 2-1, API Layers

The C language API may run on a local MPE, or there may be a version available to code running on a host processor. The low-level rendering functions must run on a local MPE. They are usually written in MPE assembly language for efficiency.

2.1.1 Low Level Direct Mode

The low level direct mode C API provides direct rendering calls. It is intended to make it easy to port applications (such as games) which do not require a high level API or which already have their own custom API. Such a custom API could be implemented in terms of the low level C API. The objects of the low level C API are primitives such as points, lines, triangles, triangle meshes, and bi-cubic patches.

2.1.2 High Level List Mode

The high-level list mode C API is designed to be easier to use, and somewhat more abstract than the direct mode API. The objects of the list mode API have attributes such as position, orientation, rotation, and movement. These objects are arranged in a hierarchy. The position and orientation of an object is defined in terms of the position and orientation of its parent. Calls are provided to render and move objects, and to test for collisions between objects. These calls work for both 2D and 3D objects, providing a unified and easy to use interface for the programmer.

2.1.3 MPE Rendering Functions

Ultimately the MPEs are used to do almost all rendering. Direct access to the MPEs allows for complete customization of the rendering process. The high-level C language API provides an easier to use interface, while still providing hooks so that the MPE rendering functions may be customized.

The MPE rendering functions are small MPE programs that take commands from the high level API and dispatch them to the MPEs being used for rendering.

The graphics APIs can use as many MPEs as are available; more MPEs will result in better performance, but sometimes some MPEs may be reserved for other uses such as audio or MPEG, and in these cases the graphics APIs will still function, but at a lower level of performance.

2.2 The Rendering Process

Calls at all layers of the 2D and 3D APIs may be mixed during the rendering process. For example, the 2D API or Movie API may be used to generate a texture to be placed on a 3D object. A 2D overlay may be placed at a constant Z value in a 3D scene, to provide a frame or titles. 3D objects may also be rendered on top of an MPEG-1 video screen, external video channel, or an image generated by the 2D API. In fact, in the high-level list mode API 2D, 3D, and movie objects are treated virtually identically, and can be intermixed freely.

User provided rendering functions might also be used to draw into the output view. If Z buffering is used, this can be done at any stage in the rendering process (subject to the restriction that transparent objects must be rendered after any objects behind them). This means that, for example, a ray tracing function may be used to render a special object into the Z buffer after (or before!) the "standard" graphics APIs have been used to draw more traditional objects such as polygons and bi-cubic patches in that buffer.

3. Basic Data Types

This section describes the basic data types and structures used by the MML2D library. All of these data types are defined within the MML2D.H header file.

3.1 System Information

The following data types maintain information about the entire NUON system.

3.1.1 mmlSysResources

The **mmlSysResources** object keeps track of system resources such as fonts, SDRAM allocation, and network sockets. The application programmer generally will only need to call the initialization function for this object and pass it to the functions that require it.

3.2 Style & Color Values

The following data types maintain information about drawing colors, line styles, and other such attributes that are used to control details about how graphics objects are rendered.

3.2.1 mmlColor

The **mmlColor** is an abstract data type hiding the specific representation of a color. Functions are supplied to create color values based upon their representations in both RGB and YCrCb color spaces.

Colors can be generated from components specified as integers in the range 0 to 255, or as floating point numbers in the range 0.0 to 1.0 (-0.5 to 0.5 for Cr and Cb).

Internally all colors are represented in integer YCrCb space that conforms to Recommendation ITU-R BT.601, which is the color space used by the hardware.

3.2.2 m2dLineStyle (not supported in SDK > 0.87)

An **m2dLineStyle** object encapsulates the style properties of a line. It is used as an argument in the line drawing functions, and a default style is included in the graphics context structure. The **m2dLineStyle** structure has the following fields:

m2dLineKind *lineKind* — The line type. There are 8 line types:

Note: types eLine5 and eLine6 can intermittently crash in SDK0.85 and some previous SDK's.

Line Type	Description
eLine1	Not anti-aliased. Straight ends.
eLine2	Not anti-aliased. Rounded ends.
eLine3	Anti-aliased. Rounded ends.
eLine4	Not anti-aliased. Rounded ends. Uses *lineRandNum*, an integer array of 4 random numbers defined in **m2dLineStyle**, to produce a speckled line.
eLine5	Anti-aliased. Rounded ends. Uses a second color, *foreColor2*, which is defined in **m2dLineStyle** to produce a 2-colored line. The line starts with *foreColor2* in the first endpoint and gradually interpolates to the first color, *foreColor*, at the second endpoint.
eLine6	Anti-aliased. Rounded ends. Uses a second color, *foreColor2*, which is defined in **m2dLineStyle** to produce a 2-colored line. The line starts with *foreColor2* in the first endpoint and gradually interpolates to the first color, *foreColor*, at the second endpoint. Has blend interpolation: in defining the endpoint colors, the last byte of each color that is normally set to zero, determines the blend level, with 0x3F being maximum opacity. So if *foreColor2* were 0xf080803f and *foreColor* were 0xf0808000, the resulting line is a solid white line that gradually becomes a faint white line. Setting the last byte of both *foreColor* and *foreColor2* to 0x00 produces no display.
eLine3clut	Emulates type **eLine3** when pixelFormat is eClut8.
eLine7clut	Emulates type **eLine5** when pixelFormat is eClut8. The line starts with *foreColor2* in the first endpoint and gradually interpolates to the first color, *foreColor*, at the second endpoint. The indices between *foreColor* and *foreColor2* will be used as pixel values, between the 2 endpoints. In other words, *foreColor* and *foreColor2* will be used as the minimum and maximum of a range of color indices. The value of *foreColor* can be less than or greater than *foreColor2*.

mmlColor *foreColor* — The line color. Line types which use a color lookup table (**eLine3clut** and **eLine7clut**) should assign an index in the table.

uint32 *thick* — The line thickness. All lines are an even number of pixels thick. An odd value of thick is automatically incremented.

uint32 *alpha* — The line translucency which is used by eLine3, eLine5 and eLine6 for anti-aliasing. Values are in the range 0 to 255, with 0 meaning completely opaque and 255 meaning completely transparent.

mmlColor *foreColor2* — This is the second color (besides *foreColor*) used by **eLine5**, **eLine6** and **eLine7clut**. Line types which use a color lookup table (**eLine7clut**) should assign an index in the table.

int32 *colorBlend1* — Both *colorBlend1* and *colorBlend2* which is described below are used by **eLine6** for blend interpolation. Uses only the lower 8 bits of int32 and eventually becomes the last byte of *foreColor*.

int32 *colorBlend2* — Used by **eLine6** for blend interpolation. Uses only the lower 8 bits of int32 and eventually becomes the last byte of *foreColor2*.

int32 *lineRandNum*[4] — This integer array of 4 random numbers is used by eLine4 to produce a speckled line.

3.2.2.1 Notes

Line types with rounded ends are extended at both endpoints by: 0.5*thick. For example, a line with endpoints [(0,10), (0,20)] and width of 10 becomes a line with endpoints [(0,15), (0,25)]. The beveled end is a half-circle with a radius of 5.

All lines are of even thickness and at least two pixels thick. The lines are approximately centered on the beginning and ending coordinates. For example, a horizontal line of type eLine1, 6 pixels thick, drawn from (10, 20) to (30, 20) would fill the rectangle from (10, 18) through (30, 23).

Lines which use a color lookup table (**eLine3clut** and **eLine7clut**) should assign the number of gradations per color in the palette to **mmlGC::nClutAlpha**. The current implementation assumes that for a certain color, a lower index value indicates a more intense value, than a higher index value. Therefore, to emulate an anti-aliased line of a certain color, create a palette where the intensity of a color decreases as the index value increases. For example, an index value of 17 for the color black is more intense than an index value of 20 for black.

Any field in the **m2dLineStyle** structure which cannot be changed by a call to *m2dInitLineStyle*() can still be changed by explicitly changing the field in the **m2dLineStyle** object. (e.g., assigning a value to *LS. foreColor2* to change the *foreColor2* field).

3.2.3 m2dEllipseStyle (not supported in SDK > 0.87)

An **m2dEllipseStyle** object encapsulates the style properties of an ellipse. It is used as an argument in the ***m2dDrawStyledEllipse*** function, and a default style is included in the graphics context structure.

The **m2dEllipseStyle** structure has the following fields:

F16Dot16 *ratio* — This fixed point number approximately represents the square root of the ratio of the ellipse width to the ellipse radius. The allowable range is from 1 to 0x1FFFE. In the case of unfilled ellipses, the width is the width of the outline. In the case of translucent filled ellipses, it is the width of the edge blur. For example, a value of 0x8000 (0.5) will yield an unfilled ellipse with an outline edge that is one fourth the thickness of the ellipse radius.

mmlColor *foreColor* — The ellipse color.

f24Dot8 *xScale* — The x-eccentricity. Assigning the same value to *yScale* generates a circle. In 8.8 format using only the lower 16 bits of f24Dot8 (i.e., the maximum value is *ox7FFF*).

f24Dot8 *yScale* — The y-eccentricity. Assigning the same value to *xScale* generates a circle. In 8.8 format using only the lower 16 bits of f24Dot8 (i.e., the maximum value is *ox7FFF*).

uint32 *alpha* — The translucency of *foreColor*. Values are in the range 0 to 255, with 0 meaning completely opaque and 255 meaning completely transparent.

int32 *fill* — Determines if an ellipse is hollow (*fill* has a value of 0) or filled (*fill* has a value of 1).

3.2.3.1 Notes

Ellipses which use a color lookup table should assign the number of gradations per color in the palette to **mmlGC::nClutAlpha**. The current implementation assumes that for a certain color, a lower index value indicates a more intense value, than a higher index value. Therefore, to emulate an anti-aliased line of a certain color, create a palette where the intensity of a color decreases as the index value increases. For example, an index value of 17 for the color black is more intense than an index value of 20 for black.

The *xScale* and *yScale* factors are multiplied by the radius provided in the ellipse drawing functions to create the x and y axes of the ellipse. The range of alpha is 0 to 255, but the 2 least significant bits are ignored.

3.2.4 mmlPoint

An **mmlPoint** is a set of three 16.16 fixed point numbers representing a point in 3D space (or in 2D space, if the Z coordinate is to be ignored).

3.2.5 m2dPoint

An **m2dPoint** is a set of two unsigned 16 bit integers representing a coordinate in 2D space.

3.2.6 m2dRect

An **mmlRect** is a set of two points representing a rectangle in 2D space. The Z coordinates of the two points are ignored.

3.2.7 m2dBox

An **m2dBox** is an unfilled rectangle (i.e. the outline of a box) that can be quickly drawn and erased.

3.2.8 mmlBox

An **mmlBox** is a set of points in 3D specifying a rectangular box.

3.2.9 mmlLayoutMetrics

An **mmlLayoutMetrics** object is used to obtain a variety of important pieces of information about the layout properties of a particular text style and font combination. This information can be used by an application to determine how text layout should be done.

The **mmlLayoutMetrics** structure has the following fields:

uint32 *columnHeight* — Pixel height of a line using this style. Equals the ascent height + descent height + between-line leading.

uint32 *base* — Number of pixels from top of line to the text baseline. (For English, this is the imaginary line on which all Capital letters rest.

f16Dot16 *ascent* — The fractional pixel height (measured from baseline) of the highest ascender in the typeface used in this style.

f16Dot16 *descent* — The fractional pixel height (measured from baseline) of the lowest ascender in the typeface used in this style. This is always a negative number.

f16Dot16 *maxWidth* — The fractional pixel width of the widest glyph in the typeface used in this style.

uint32 *ttPointSize* — The actual pointsize used by the T2K type engine to render the text in this style.

uint32 *firstCharCode* — The lowest character code represented in the font used for this style.

uint32 *lastCharCode* — The highest character code represented in the font used for this style.

uint32 *numCharacters* — The total number of characters represented in the font used for this style.

3.2.10 Miscellaneous Enumerated Types

In addition to the data types mentioned above, there are a number of enumerated types. They are used to specify unique parameter values used as arguments by various library functions, or as fields of a larger data structure. Some of these types are listed in the table below.

charKind	eBool	m2dFill
m2dLineKind	m2dTextEmphasis	m2dTextWeight
mmlDisplayFlags	mmlDrawOp	mmlPixAspect
mmlPixFormat	mmlPixmapProperties	MmlStatus
mmlVideoLayer	mmlZCompare	TextEncoding
textMix	typeTechnology	eSafeColorSel

All of these enumerated types, along with the corresponding definitions of specific values, are found in the MML2D.H include file.

3.3 Graphics Context

3.3.1 mmlGC

The information about how to draw 2D graphics primitives is stored in a *graphics context* structure of type **mmlGC**. This object encapsulates state for the rendering functions.

Most functions have a large number of options; for example, the width of lines to be drawn, the color to be used in filling areas, whether aspect ratio correction is on or off, etc. Rather than passing each option as an individual parameter to every drawing function, we keep the state in a graphics context and pass the drawing function a pointer to that structure. This also makes it possible to set up a global state that applies to subsequent drawing commands.

The graphics context structure has two main purposes: one is to simplify the interfaces to drawing functions (which would otherwise need a large number of parameters to specify how to draw primitives), and the other is to make it convenient to draw a series of primitives with the same style and settings.

For example, a program might keep one graphics context for drawing the main region of its display in solid colors, and a second for use in an anti-aliased translucent overlay on top of the main screen.

The **mmlGC** structure has the following members:

3.3.1.1 mmlGC::z

Fixed16 *z*

A 16.16 fixed point number giving a default Z value to be used in Z comparisons in the event that the source data has no Z buffer.

3.3.1.2 mmlGC::alpha

uint32 *alpha*

The least significant 16 bits are used as the alpha value in functions that can do translucent drawing. A value of 0xFFFF is completely opaque. A value of 0x0000 is completely transparent.

3.3.1.3 mmlGC::foreColor

mmlColor *foreColor*

Gives the default color to use when drawing; this color will be used in any operations that do not explicitly specify a color. The alpha value associated with the color, if any, will be ignored; the **alpha** member of the **mmlGC** structure will determine the alpha value to be used when this is required.

3.3.1.4 mmlGC::backColor

mmlColor *backColor*

Gives the default background color when drawing. Many graphics objects such as text characters and hollow shapes have interior portions. These interiors will be drawn with the color specified by the ***backColor*** field. If this color is set to **eTransparent**, the interior pixels are treated as transparent and are not changed.

3.3.1.5 mmIGC::nClutAlpha

uint32 *nClutAlpha*

The number of indices in a palette that will be used to draw a line or ellipse in color lookup table mode. This is used if the pixel format of the destination pixmap is **eClut4** or **eClut8**. This is equivalent to the number of gradations of a color that will be used by the drawing routine.

3.3.1.6 mmIGC::clutForeIndex

uint32 *clutForeIndex*

Serves the same function as the **foreColor** field when the destination pixmap has a pixel format of **eClut4** or **eClut8**.

3.3.1.7 mmIGC::clutBackIndex

uint32 *clutBackIndex*

Serves the same function as the **backColor** field when the destination pixmap has a pixel format of **eClut4** or **eClut8**.

3.3.1.8 mmIGC::fixAspect

ebool fixAspect

fixAspect value	Description
eFalse	Causes copying to be done in true display pixels. But the width to height ratio of these pixels is 8:9, so a square drawn as 9 pixels by 9 pixels will appear to the viewer as a rectangle.
eTrue	Causes copying to be done in square pixels. The edge of each of these pixels has a length of one scan line height. So a square drawn as 8x8 will occupy a rectangle 9 pixels wide by 8 pixels high, which will appear to the viewer as square.

3.3.1.9 mmIGC::defaultLS

m2dLineStyle *defaultLS*

Provides the default line style used in line drawing functions. The application may override this default by using the ***m2dDrawStyled2Dline*** function.

3.3.1.10 mmlGC::defaultES

m2dEllipseStyle *defaultES*

Provides the default ellipse style used by the functions ***m2dDrawEllipse*** and ***m2dDrawQuadArc***. The application may override this default by using the ***m2dDrawStyled**** functions.

3.3.1.11 mmlGC::disCopyBlend

uint32 *disCopyBlend*

Controls blending when using the ***m2dCopyRectDis*** function. If set to 0, the copy is a simple source copy. If set to 1, the source will be blended with the destination pixels using the 8 bit alpha value of each source pixel. This only happens if the source pixel format is **e888Alpha**.

3.3.1.12 mmlGC::textBase

uint32 *textBase*

When the **textMix** parameter of a text output function is set to **eClutAlpha**, the **textBase** field of the graphics context contains the CLUT index used for the most translucent text pixels. All text pixel values are added to **textBase** to obtain the index used in the CLUT.

3.3.1.13 mmlGC::textDiv

uint32 *textDiv*

Text pixel values normally range between 0 and 63 from most transparent to most opaque. When the **textMix** parameter of a text output function is set to **eClutAlpha**, the text pixel values are divided by **textDiv** and then added to **textBase** to get an index into the CLUT. Satisfactory text can generally be obtained using only 4 or 8 CLUT entries, so typical values of **textDiv** would be 16 or 8.

3.3.1.14 mmlGC::textMin

uint32 *textMin*

When the **textMix** parameter of a text output function is set to **eClutAlpha**, the CLUT index used when text is drawn is clipped to a minimum value specified by **textMin**.

3.3.1.15 mmIGC::textMax

uint32 *textMax*

When the **textMix** parameter of a text output function is set to **eClutAlpha**, the CLUT index used when text is drawn is clipped to a maximum value specified by **textMax**.

3.3.1.16 mmIGC::translucentText

uint32 *translucentText*

Normally set to 0. Set to 1 if text is to be displayed that is translucent with respect to the underlying video plane. This provides proper alpha channel antialiasing when the foreground and/or background colors are translucent.

3.3.1.17 mmIGC::textWidthScale

F16Dot16 *textWidthScale*

Represents the ratio of the width of characters to the height. A value of 0x10000 (default value of 1.0) provides the aspect ratio intended by the type designer, presuming square pixels on the output device.

Different values can be used to create condensed or expanded text, or to compensate for non-square pixels on the output device.

3.3.1.18 mmIGC::transparentOverlay

eBool *transparentOverlay*

If set to **eTrue**, copy functions can create transparent pixels in the overlay plane. See the description of the copy functions.

3.3.1.19 mmIGC::transparentSource

eBool *transparentSource*

If set to **eTrue**, copy functions will not copy designated pixels to the target display pixmap. See the description of the copy functions.

3.3.1.20 mmlGC:: rgbTransparentValue

int *rgbTransparentValue*

A 15 bit RGB color that will be treated as transparent or non-copying if the **transparentOverlay** or **transparentSource** flags are set to **eTrue**. See the description of the copy functions.

3.4 Bitmap Definitions

The basic bitmap definition used by NUON is known as a *pixmap*.

The **mmlPixmap** structure defines all of the attributes of a bitmapped image on NUON. There are two other types which should be considered as subclasses: **mmlAppPixmap** and **mmlDisplayPixmap**.

Within the MML2D.H include file, you'll find that the definitions for **mmlPixmap**, **mmlAppPixmap**, and **mmlDisplayPixmap** are essentially identical. This is because the main difference between these types is not something that can be specified as part of a structure definition.

The difference between an **mmlAppPixmap** and an **mmlDisplayPixmap** is simply where the corresponding bitmap is located in memory. This is a very important distinction, however, because it affects what hardware operations may be performed on the bitmap.

It would have been possible to use the basic **mmlPixmap** type for all bitmap descriptions, but this would have required many library functions, and likely your own application code, to include a lot of additional code to determine and/or verify the type of bitmap being passed as a parameter.

By using separate type definitions for each type of bitmap, we gain the ability to do compile-time error checking. We allow the library, or your application's own code, to specify to the C/C++ compiler that a particular function expects a specific type of bitmap. This way, for example, if you pass the ***mmlSimpleVideoSetup*** function a pointer to an **mmlAppPixmap** when it expects an **mmlDisplayPixmap**, the compiler can throw out an error message.

3.4.1 mmlPixmap

The **mmlPixmap** structure is a generic bitmap description. The bitmap described is also either an **mmlAppPixmap** or **mmlDisplayPixmap** (but not both at once).

There are times when a function is capable of dealing with either type of bitmap. In those cases, the more generic **mmlPixmap** type may be used in place of one of the more specific types.

3.4.2 mmlAppPixmap

The **mmlAppPixmap** structure defines an application pixmap. Application pixmaps are always located in system RAM and can be addressed directly (through the cache) via ***(pixmapP->memP)** or by using Other Bus DMA.

Application pixmaps are laid out in a traditional raster fashion, and therefore may be easily manipulated directly by an application. However, they cannot be directly displayed by the NUON's video display hardware or used for bilinear DMA transfers.

A major purpose of an application pixmap is to act as a source for a block-transfer to a display pixmap.

3.4.3 mmlDisplayPixmaps

The **mmlDisplayPixmap** structure defines a display pixmap. Display pixmap memory must be allocated in Merlin SDRAM. They may be accessed via Main Bus DMA or used as framebuffers for the NUON's video hardware.

If you attempt to access a display pixmap directly (through the cache) via ***(pixmapP->memP)**, you will not get sensible results because the memory layout does not follow a traditional raster format. The format used is designed for efficient access by the DMA and video display hardware. The layout is sufficiently convoluted that it cannot be efficiently addressed directly via software.

Instead of accessing the pixmap memory directly, applications must use DMA. The NUON's Main Bus DMA system has special pixel-mode access functions for accessing arbitrary pixels or rectangular regions of a display pixmap.

3.5 Command Sequences

3.5.1 mmlSequence

An **mmlSequence** object describes a list of drawing operations that can be executed as a single command, and executed repeatedly as needed. See chapter 13 for more information.

This page intentionally left blank.

4. General Media Library Functions

4.1 Pixmap Initializers and Attribute Setting

Different data types are used for the two subclasses of pixmap, so that compilers can do type checking in the functions that only accept one of the subclasses as a parameter; for example:

```
ScaledCopyRect( mmlAppPixmap* srcP, mmlDisplayPixmap* destP, ...)
```

Any pixmap may be correctly passed to a function that specifies **mmlPixmap** as a parameter by casting the subclass to the superclass, e.g.

```
mmlAppPixmap map1;
mmlInitAppPixmaps( &map1, 640, 480, NULL, e8Clut, 1 );
mmlSetPixMapClut( (mmlPixmap*)&map1, clutPtr );
```

The pixmap initializers allow the programmer to either specify an already allocated area of memory to be used for the pixmap memory, or to have the initializer function allocate the memory. If the memory is automatically allocated, the programmer should call **ReleasePixmaps** to free the memory when the pixmap is no longer used. Provision is made for initializing an array of pixmaps of the same type with a single call. When the pixel type is e655Z, this actually results in memory being allocated for a Z buffer that is shared by the color buffers. In this case, the programmer must make sure to Release the array of pixmaps with a single **ReleasePixmaps** call that points to the first pixmap in the array.

Warnings — It is illegal to attempt to initialize a Display Pixmap with an address that is not in Merlin SDRAM. However, it is possible to initialize an Application Pixmap with an SDRAM address. If the application code can be guaranteed to always run on NUON (never on the host side of combination Host-Merlin platforms), then the SDRAM Application pixmap can be addressed directly via `*(PixmapP->memP)`, but it is considered poor practice to create such non portable code.

4.1.1 mmlInitAppPixmaps

mmlStatus mmlInitAppPixmaps(mmlAppPixmap* sP, mmlSysResources* srP, int wide, int high, mmlPixFormat pixFormat, int numBuffers, void* memP);

Creates an array of one or more appPixmap objects of dimensions *wide* * high pixels, using the specified *pixFormat*.

pixFormat must be one of the following enumerated values:

pixFormat value	Description

pixFormat value	Description
eClut4	4 bits per pixel with color lookup. Values are indices into a table of 16 YCC values each in **e888Alpha** format.
eClut8	8 bits per pixel with color lookup. Values are indices into a table of 256 YCC values each in **e888Alpha** format.
e655	16 bits per pixel, with 6 bits for the Y channel, and 5 bits for each of Cr and Cb.
e888Alpha	32 bits per pixel, with 8 bits for each of Y, Cr, and Cb and 8 bits for alpha.
e655Z	32 bits per pixel, 16 for YCrCb color and 16 for Z buffer depth.
e888AlphaZ	64 bits per pixel, 24 for YCrCb color, 8 for alpha, and 32 for Z depth.
eRGBAlpha1555	16 bits per pixel, 1 bit alpha, 5 each for R, G, and B components
eRGB0555	16 bits per pixel, most significant bit is 0, 5 each for R, G, and B components.

numBuffers can be 1, 2, or 3. This specifies the number of pixmaps that will be initialized.. Generally, only 1 application pixmap will be initialized with this call.

4.1.2 mmlInitDisplayPixmaps

mmlStatus mmlInitDisplayPixmaps(mmlDisplayPixmap* sP, mmlSysResou*rces* srP, int wide, int high* , mmlPixFormat pix, int numBuffers, void* memP);

Initializes an array of one or more displayPixmap objects of dimensions *wide* * *high* pixels, using the specified *pixFormat*.

pixFormat must be one of the YCC formats in the previous table. If the pixmap will be displayed in the main video channel rather than the graphics overlay channel, the *pixFormat* can not be **eClut4** or **eClut8**.

Wide must be a multiple of 8 pixels. For format eClut8, wide must be a multiple of 16 pixels, and for format eClut4, wide must be a multiple of 32 pixels.

numBuffers can be 1, 2, or 3, and has special meaning when used with *pixFormat* **e655Z**. In this format, multiple buffers share a common Z buffer, so it is possible while Buffer A is being displayed to do Z rendering into Buffer B, clear the Z buffer and start rendering into Buffer C. When Buffer A has finished displaying, Buffer B can start displaying. Because the Z buffer is shared, it is not possible to separately do Z buffering into two buffers at the same time.

4.1.3 mmlReleasePixmaps

void mmlReleasePixmaps(mmlPixmap* sP, mmlSysResources* srP);

Release an array of one or more previously initialized pixmaps. If an array of pixmaps was initialized by a single call, they should be released by a single call.

4.1.4 mmlSetPixmapClut

void mmlSetPixmapClut(mmlPixmap* sP, mmlColor* clutPtr);

Set the color lookup table used by a pixmap to the table pointed at by *clutPtr*. A color lookup table is simply an array of 256 **mmlColor** values. It should be aligned on a 1024 byte boundary.

* In previous versions, this function has been called ***mmlSetClut***.

4.2 Other Initialization

4.2.1 mmlPowerUpGraphics

void mmlPowerUpGraphics(mmlSysResources* sysResPtr)

This function should be called at the beginning of every application, to initialize the *sysRes* object.

4.2.2 mmlInitGC

void mmlInitGC(mmlGC* gcPtr, mmlSysResources* sysResPtr)

This function should be called at the beginning of every application, following the ***mmlPowerUpGraphics*** call. It sets default values for the graphics context that can then be changed by the application program.

4.2.3 mmlSimpleVideoSetup

void mmlSimpleVideoSetup(mmlDisplayPixmap* sp, mmlSysResources* srP, mmlVideoFilter filttype);

In the current version of the mmlibrary, this function can be used to initalize a video function that will cause the displayPixmap to be displayed. It can only be used for a single pixmap.

The *filttype* parameter can be one of **eNoVideoFilter**, **eTwoTapVideoFilter**, or **eFourTapVideoFilter**. The two tap filter will generally provide adequate anti-flicker filtering for interlaced TV.

4.2.4 mmlInitFontContext

mmlInitFontContext(mmlGC *gcPtr***, mmlSysResources*** *srP***, mmlFontContext*** *fcPtr***, int** *cacheSize* **);**

This function should be called at the beginning of every application that uses fonts, following the ***mmlInitGraphicsContext*** call. It sets default values for the font context that can then be changed by the application program. The *cacheSize* parameter limits the amount of memory that will be allocated for storing cached glyph pixmaps.

4.2.5 m2dSetPoint

m2dPoint m2dSetPoint(uint16 *x***, uint16** *y* **);**

Returns an **m2dPoint** structure with components x and y.

4.2.6 m2dSetRect

m2dRect* m2dSetRect(m2dRect* *rP***, uint16** *left***, uint16** *top***, uint16** *right***, uint16** *bottom* **);**

Returns a pointer to an initialized **m2dRect** structure with components describing the left top point of the rectangle and the right bottom point of the rectangle.

5. Color Allocation and Manipulation

The MML library has functions to construct an abstract color (type **mmlColor**) from three components, either RGB or YCrCb, with each component expressed either in the integer range 0 to 255 or in the floating point range 0.0 to 1.0, depending on the function. With one exception, the functions that return an **mmlColor** make sure that the color is valid and safe. A valid color is one that conforms to Rec. ITU-R BT.601, the international standard for digital video. This standard limits the range of the luma and chroma components to a range somewhat less than the full dynamic range of their numeric representations:

16 <= *Y* <= 235
16 <= *Cr* <= 240
16 <= *Cb* <= 240

A safe color is one that can be correctly displayed on an NTSC or PAL composite video device such as a television. Even when restricted to the Rec. ITU-R BT.601 limits, not all component triples generate colors that can be displayed on the NUON video device. In particular, the gamut of colors displayable on a composite video monitor (NTSC or PAL) is restricted by requirements on the strength of the luminance and chrominance signals: the sum of the two signals cannot be too large or too small, or the monitor will be unable to show it. Some older monitors and TV receivers can even lose sync if the color signal is a supersaturated blue. The restrictions on colors shown on S-video or component video monitors are much less stringent. Therefore a color that is safe for NTSC or PAL composite video is also safe for S-video and component video.

When the arguments to a color-generating function produce a color that is unsafe, the function modifies it: the luminance and hue are preserved, but the color is de-saturated by effectively adding white to it. The advanced user can control how unsafe colors are modified through the **mmlSafeColorLimits()** function.

5.1.1 mmlColorFromRGB

mmlColor mmlColorFromRGB(uint8 *red*, **uint8** *green*, **uint8** *blue*)

Creates a color based upon an RGB specification. The *red*, *green*, and *blue* arguments each can be in the range 0 to 255. Values less than 0 will be forced to 0; values greater than 255 will be forced to 255. The control component of the return value is zero.

5.1.2 mmlColorFromYCC

mmlColor mmlColorFromYCC(**uint8** *Y*, **uint8** *Cr*, **uint8** *Cb*)

Creates a color based upon a YCrCb specification. The *Y*, *Cr*, and *Cb* arguments can each be in the range 0 to 255. *Cr* and *Cb* values of 128 correspond to neutral. Values less than 0 will be forced to 0; values greater than 255 will be forced to 255. The control component of the return value is zero.

NOTE: This function simply packages the specified component values into a **mmlColor** type and returns it. It does not force the component arguments to lie within the ranges imposed by Rec. ITU-R BT.601. Nor does it restrict the resulting color to be safe for composite video display. This function is the only exception: all other color-producing functions in the library return valid and safe colors. To insure a valid and safe color, pass the result to the function **mmlSafeColor()**.

5.1.3 mmlGetYCCComponents

void mmlGetYCCComponents(**mmlColor** *col*, **uint8** **Y*, **uint8** **Cr*, **uint8** **Cb*)

Deconstructs the YCrCb representation of the abstract color *col*. Argument *col* does not have to be a safe color.

5.1.4 mmlColorFromYCCf

mmlColor mmlColorFromYCCf(**double** *Y*, **double** *Cr*, **double** *Cb)*

Creates a color based upon a YCrCb specification. Luminance argument *Y* must be in the range 0.0 to 1.0. Chrominance arguments *Cr* and *Cb* must each lie in the range -0.5 to 0.5. Values outside these limits are forced to the nearest limit. The control component of the return value is zero.

5.1.5 mmlGetYCCFloatComponents

void mmlGetYCCFloatComponents(**mmlColor** *col*, **double** **Y*, **double** **Cr*, **double** **Cb*)

Extracts floating point YCC components from the **mmlColor** value specified by the *col* parameter. Argument *col* does not have to be a safe color.

5.1.6 mmlGetYCCComponents

void mmlGetYCCComponents(**mmlColor** *col*, **uint8** **Y*, **uint8** **Cr*, **uint8** **Cb*)

Extracts YCC components from the **mmlColor** value specified by the *col* parameter. Argument *col* does not have to be a safe color.

5.1.7 mmlColorFromRGBf

mmlColor mmlColorFromRGBf(double *rf*, **double** *gf,* **double** *bf*)

Creates a color based upon a floating point RGB specification. Arguments *rf*, *gf*, and *bf* must each lie in the range 0.0 to 1.0. Values outside these limits are forced to the nearest limit. The control component of the return value is zero.

5.1.8 mmlGetRGBFloatComponents

void mmlGetRGBFloatComponents(**mmlColor** *col*, **double** **rf*, **double** **gf*, **double** **bf*)

Extracts floating point RGB components from the **mmlColor** value specified by the *col* parameter. Argument *col* does not have to be a safe color.

5.1.9 mmlGetRGBComponents

void mmlGetRGBComponents(**mmlColor** *col*, **uint8** **rf*, **uint8** **gf*, **uint8** **bf*)

Extracts RGB components from the **mmlColor** value specified by the *col* parameter. Argument *col* does not have to be a safe color.

5.1.10 mmlSafeColor

mmlColor mmlSafeColor (**mmlColor** *col*)

Converts abstract color *col* into a valid and safe color for the platform's display. The 8-bit color or alpha component is returned unchanged.

5.1.11 mmlSafeColorLimits

int mmlSafeColorLimits (**int** *select*)

Chooses the set of limits that are applied to modify unsafe colors into safe colors. Argument *select* can be any of the values shown in the following table.

select	Meaning
eSafeColorDefault	Use the default color limits as specified by the platform.
eSafeColorNTSC	Use color limits appropriate for NTSC with a 7.5 percent setup.
eSafeColorNTSCZero	Use color limits appropriate for NTSC with zero setup
eSafeColorPAL	Use color limits appropriate for PAL (no setup).
eSafeColorCustom	Use the custom set of limits as most recently set by a call to **mmlCustomSafeColorLimits()** (see below).
eSafeColorDisable	Disable the conversion of unsafe colors into safe colors. However, colors are still modified to be valid.

The function returns 1 if the argument is valid or 0 if the argument is not valid.

5.1.12 mmlCustomSafeColorLimits

int mmlCustomSafeColorLimits (**double** *ped*, **double** *smax*, **double** *smin*, **double** *cmax*)

The library provides built-in safe-color limits that are suitable for almost every application. But it also provides the means for the adventurous and knowledgeable designer to supply custom limits.

The function's four arguments set the color limits, as described below. All four arguments are expressed in IRE units.

ped Specifies the setup or pedestal. This value is usually 0.0 or 7.5 for NTSC and 0.0 for PAL, but it can be anything. The default is 7.5 for NTSC and 0.0 for PAL.

smax Specifies the maximum allowed value for the composite video signal *Y*+*C*. If the composite signal exceeds `smax`, then the chroma *C* is reduced so that the limit is satisfied. The default value is 110 IRE for both NTSC and PAL.

smin Specifies the minimum allowed value for the composite video signal *Y*–*C*. If the composite signal falls below `smin`, then the chroma *C* is reduced so that the limit is satisfied. The default value is –15 IRE for both NTSC and PAL.

cmax Specifies the maximum amplitude permitted for the chroma excursion. That is, if the chroma is greater than `cmax`, then it is reduced to `cmax`. The

default value is 50 IRE for both NTSC and PAL.

The function not only sets new custom safe-color limits, it also selects **eSafeColorCustom** as the current limit set. It returns 1 if the arguments are valid, or 0 otherwise.

This page intentionally left blank.

6. Copy Functions

The MML copy functions provide fast copying of rectangular regions within the same pixmap or between pixmaps. Source and destination images can be combined with alpha blending.

6.1.1 m2dReadPixels

void m2dReadPixels(mmlGC ***gc**, **uint32** ***buffer**, **mmlDisplayPixmap** ***srcPtr**, **int** *x*, **int** *y*, **int** *num*, **eBool** *verticalQ*)

Reads a line of up to 64 pixels from the display pixmap beginning at the coordinates (x,y). If *verticalQ* is *true*, it reads a vertical line, else a horizontal line. The pixels are packed into the specified buffer as raw data. 16-bit pixels are packed 2 per long, 8-bit pixels are packed 4 per long, and so on. The buffer must have been allocated large enough to receive the data. The *num* argument specifies the number of pixels to be transferred and must be no greater than 64. No bounds checking is done. For 16 bit pixels, num must be a multiple of 2. For 8 bit pixels, num must be a multiple of 4. For 8 bit pixels, if a vertical line is read, it will be 2 pixels wide, and num/2 pixels high.

Not implemented for 4 bit pixels.

6.1.2 m2dWritePixels

void m2dWritePixels(mmlGC ***gc**, **uint32** ***buffer**, **mmlDisplayPixmap** ***srcPtr**, **int** *x*, **int** *y*, **int** *num*, **eBool** *verticalQ*)

Writes a line of up to 64 pixels from the specified buffer into the display pixmap beginning at the coordinates (x,y). If *verticalQ* is true, it writes a vertical line, else a horizontal line. The pixels are packed into the buffer as raw data. 16-bit pixels are packed 8 per long, 8-bit pixels are packed 4 per long, and so on. The *num* argument specifies the number of pixels to be transferred and must be no greater than 64. The line of pixels must fit within the pixmap boundaries. No bounds checking is done. For 16 bit pixels, num must be a multiple of 2. For 8 bit pixels, num must be a multiple of 4. For 8 bit pixels, if a vertical line is written, it will be 2 pixels wide, and num/2 pixels high.

Not implemented for 4 bit pixels.

6.1.3 m2dCopyRect

void m2dCopyRect(mmlGC **gc*, mmlAppPixmap **srcPtr*, mmlDisplayPixmap **dstPtr*, m2dRect* *rPtr*, m2dPoint *destLocation*)

Copies a rectangle from a source **mmlAppPixmap** to a rectangle in a destination **mmlDisplayPixmap**. The **m2dPoint** *destLocation* provides the left and top coordinates of the destination rectangle. If the source and destination use different color spaces, or pixel formats, conversion will be done. If the **mmlGC** field *fixAspect* is set to **eTrue**, horizontal aspect ratio scaling of 8:9 is done during the copy. The destination rectangle must be wide enough to contain the extra pixels.

RESTRICTION: The destination pixmap can not have the **eClut4** or **eClut8** pixelFormat..

6.1.4 m2dCopyClutRect

void m2dCopyClutRect(mmlGC* *gcP*, mmlPixmap* *srcP*,
mmlDisplayPixmap* *destP*, m2dRect* *rP*,
m2dPoint *pt*);

Copies a rectangle from a source clut-based **mmlPixmap** to a rectangle in a clut-based destination **mmlDisplayPixmap**. The **m2dPoint** *destLocation* provides the left and top coordinates of the destination rectangle. The source pixmap may be either an **mmlAppPixmap** or an **mmlDisplayPixmap**. The rectangle edges and destination coordiates may be on any pixel boundary.

RESTRICTION: Both pixmaps must have the format **eClut8**. No scaling is done.

6.1.5 m2dCopyRectScaled

void m2dCopyRectScaled(mmlGC **gc*, mmlAppPixmap **srcPtr*, mmlDisplayPixmap **dstPtr*, m2dRect* *srcRecPtr*, m2dRect* *destRecPtr,* int *horNum,* int *horDen,* int *verNum,* int *verDen*)

Scales and copies a rectangle from a source **mmlAppPixmap** to a rectangle in a destination **mmlDisplayPixmap**. The **m2dRect** *destRect* provides the left and top coordinates of the destination rectangle. If the source and destination use different color spaces, or pixel formats, conversion will be done. The horizontal scaling factor is *horNum*/*horDen* and the vertical scaling factor is *verNum*/*verDen*. The **mmlGC** field *fixAspect* is ignored. The right and bottom of the scaled rectangle are clipped to the destination rectangle.

RESTRICTION: The destination pixmap cannot have the **eClut4** or **eClut8** pixelFormat.

6.1.6 m2dCopyRectFast

void m2dCopyRectFast(mmlGC **gc*, **mmlAppPixmap** **srcPtr*, **mmlDisplayPixmap** **dstPtr*, **m2dRect*** *rPtr*, **m2dPoint** *destLocation*)

Copies a rectangle from a source **mmlAppPixmap** to a rectangle in a destination **mmlDisplayPixmap**. The **m2dPoint** *destLocation* provides the left and top coordinates of the destination rectangle. This is the fastest form of copyRect (more than 10 times faster in some circumstances), but it has many restrictions. No color conversion is done, and no scaling or aspect-ratio conversion is done.

Restrictions: The pixel formats of the source pixmap and the destination pixmap must be the same, except it is allowable for the source pixmap format to be e888Alpha while the destination format is e655. (The formats must be YCrCb formats, not RGB). The source rectangle must be complete, it can not be NULL. The left edge of the source rectangle must be on an even byte (automatically true except for eClut8 format). The width of the rectangle must be a multiple of 4 bytes (always true for e888Alpha format, but e655 formats must have a width that is an even number of pixels, and eClut8 formats must have a width that is a multiple of 4 pixels. The left edge of the destination point must be aligned on a multiple of 4 bytes (always true for e888Alpha format, but e655 format destinations must have an even x value, and eClut8 format destinations must be a multiple of 4 pixels.

6.1.6.1 Transparent Pixels in Graphics Overlay Plane

Unscaled copies from an application pixmap to a display pixmap that will be displayed in the Graphics Overlay plane can designate transparent pixels. Transparent pixels allow the underlying video plane to be seen rather than the overlay pixel. The value of the overlay pixel is completely ignored. If the pixel format of the display pixmap is e655 or e888Alpha and the graphics context value *transparentOverlay* is set to **eTrue**, then transparent pixels are specified as follows.

Source format is eRGBAlpha1555

If the application pixmap format is eRGBAlpha1555, any pixel with the most significant bit set to 1, will be translated into a transparent pixel. Other pixels will be converted to YCrCb values and displayed as opaque pixels.

Source format is eRGB0555

If the application pixmap format is eRGB0555, any pixel whose rgb value is equal to the graphics context value *rgbTransparentValue* will be translated into a transparent pixel. Other pixels will be converted to YCrCb values and displayed as opaque pixels. *rgbTransparentValue* must be a 15 bit value, i.e. the MSB must be 0.

Source format is e888Alpha

If the application pixel format is **e888Alpha** and the overlay display pixmap format is **e655**, any pixel with an alpha value of 0xFF will be translated into a transparent pixel. Other pixels will be truncated from 888YCC to 655YCC and displayed as opaque pixels. ***Not implemented in version 2.0**

If the application pixel format is **e888Alpha** and the overlay display pixmap format is also **e888Alpha**, the *transparentOverlay* flag is ignored. Pixels are copied directly, and the alpha value governs the opacity of the overlay plane pixels; alpha values of 0xFF are completely transparent to the underlying Video plane, and alpha values of 0 are completely opaque.

When the overlay display pixmap format is **e655** or **e888Alpha**, any pixel with a value of 0 will be completely transparent to the video plane. When the display format is **e655**, this means that any **e888Alpha** pixel with a value between (0,0,0) and (3,7,7) will automatically become a transparent pixel regardless of the alpha value. When these pixels are copied to a display pixmap with **e888Alpha** format, pixels with a value of 0,0,0 will still be transparent, but other small values will have a green color.

Note that RGB values never translate to a 0,0,0 YCC value, so no RGB pixel is ever accidentally translated into a transparent pixel.

Restrictions: Transparent overlay pixels require: the copy must be unscaled; the display pixmap format must be **e888Alpha** or **e655**; the display pixmap must be displayed in the Graphics Overlay plane; and the graphics context value *transparentOverlay* must be set to **eTrue**;

6.1.6.2 Pixels That Do Not Transfer

When the *transparentOverlay* flag is false, a different form of copy transparency is available. Unscaled copies from an application pixmap to a display pixmap with pixel format e655 or e888Alpha, can designate pixels that will not be copied. The target pixmap is left unchanged for these pixels. This mode can be used for display pixmaps that are displayed in either the graphics overlay plane, or the main video plane. If the graphics context value *transparentSource* is set to eTrue, non-copying pixels are designated as follows:

Source format is eRGBAlpha1555

If the application pixmap format is eRGBAlpha1555, any pixel with the most significant bit set to 1, will not be copied. Other pixels will be converted to YcrCb values and copied to the display pixmap.

Source format is eRGB0555

If the application pixmap format is eRGB0555, any pixel whose rgb value is equal to the graphics context value *rgbTransparentValue* will not be copied. Other pixels will be converted to YcrCb values and copied to the display pixmap.

rgbTransparentValue must be a 15 bit value, i.e. the msb must be 0. ***Not implemented in version 2.0**

Source format is e888Alpha

If the *transparentOverlay* value is eFalse and the *transparentSource* value is set to eTrue, then pixels with an alpha value of 0xFF will not be copied.

Restrictions: Transparent source pixels require: the copy must be unscaled; the display pixmap format must be e888Alpha or e655; the graphics context value *transparentOverlay* must be set to eFalse, and the graphics context value *transparentSource* must be set to eTrue.

6.1.7 m2dCopyRectDis

void m2dCopyRectDis(mmlGC ****gc*, **mmlDisplayPixmap ****srcPtr*, **mmlDisplayPixmap ****dstPtr*, **m2dRect* *rPtr*, **m2dPoint** *destLocation*)

Copies a rectangle from a source **mmlDisplayPixmap** to a rectangle in a destination **mmlDisplayPixmap**. The **m2dPoint** *destLocation* provides the left and top coordinates of the destination rectangle.

This page intentionally left blank.

7. Fill Functions

7.1.1 m2dDrawPoint

void m2dDrawPoint(mmlGC * *gc*, **mmlDisplayPixmap *** *V*, **int** *x*, **int** *y,* **mmlColor color**)

Draws a single point at coordinates (*x*, *y*) in a **mmlDisplayPixmap** *V*, using the specified *color*. The pixmap must use 16 bit or 32 bit pixels. The alpha component of the color will be copied if the **mmlDisplayPixmap** uses 32 bit pixels. NOTE: This is actually implemented as a macro and will always be the fastest implementation for plotting a single point in a display pixmap.

7.1.2 m2dSmallFill

void m2dSmallFill(mmlGC * *gc*, **mmlDisplayPixmap *** *V*, **int** *x*, **int** *y,* **mmlColor color, int** *xLen*, **int** *yLen*)

Fills a small rectangle with a solid color. The pixmap must use 16 bit or 32 bit pixels. The alpha component of the color is ignored. The dimensions of the rectangle are given by *xLen* and *yLen*, and the product *xLen * yLen* must be no greater than 64 pixels. NOTE: This is actually implemented as a macro and will always be the fastest implementation for filling a small line or rectangle in a display pixmap.

7.1.3 m2dFillColor

mmlColor m2DFillColor(**mmlGC*** *gcP*, **mmlDisplayPixmap*** *V*,
m2dRect* *rP*, **mmlColor** *color*)

Fill a rectangle in a **mmlDisplayPixmap** with a color. The rectangle must be within the bounds of the pixmap. This function can be used to plot individual points and to draw horizontal and vertical lines.

The ***m2dFillClut*** function should be used for clut-based pixmaps.

7.1.4 m2dFillClut

void m2dFillClut(mmlGC* *gcP*, **mmlDisplayPixmap*** *destP*,
m2dRect* *rP*, **mmlColor** *color* **);**

Fill a rectangle in a clut based **mmlDisplayPixmap** with a color index. The rectangle must be within the bounds of the pixmap. This function can be used to plot individual points and to draw horizontal and vertical lines.

The rectangle edges can be on any pixel boundary. The *color* used as a fill is not an **mmlColor**, but is an index into the CLUT. In this case, *color* should be of the form:

```
(index<<24) | (index<<16) | (index<<8) | index
```

8. Line Draw Functions

8.1.1 m2dInitLineStyle (not supported in SDK > 0.87)

void m2dInitLineStyle(**mmlGC*** gcP, **m2dLineStyle*** lineS, **mmlColor** color, **int32** thick, **int32** alpha,**m2dLineKind** lineKind)

Call to initialize a **m2dLineStyle** object with a specific *color*, *thickness*, *alpha,* and *line type* value. The **m2dLineStyle** structure is described in detail in section 3.2.2.

8.1.2 m2dDrawLine (not supported in SDK > 0.87)

void m2dDrawLine(**mmlGC** **gc*, **mmlDisplayPixmap** **V*, **int** *startx*, **int** *starty*, **int** *endx*, **int** *endy*)

Draws a line from coordinates (*startx,starty*) to (*endx,endy*) inclusive. The line thickness, color, drawing mode, z comparison mode, alpha value (if applicable), and z value (if applicable) are all taken from the graphics context *gc*. The default line type is eLine3 which is anti-aliased.

NOTE: Any of the fields in the *gc->defaultLS* object can be explicitly changed. This will remain the setting for future line drawing until the value is again changed..

8.1.3 m2dDrawStyledLine(not supported in SDK > 0.87)

void m2dDrawStyledLine(**mmlGC** **gc*, **mmlDisplayPixmap** **V*, **mmlLineStyle** **stylePtr*, **int** *startx*, **int** *starty*, **int** *endx,* **int** *endy*)

Draws a line from coordinates (*startx,starty*) to (*endx,endy*) inclusive. The line thickness, color, and other style properties are taken from the **mmlLineStyle** object. These properties override the properties in *gc->defaultLS*, but do not replace them.

8.1.4 m2dDrawPolyLine (not supported in SDK > 0.87)

void m2dDrawPolyLine(**mmlGC** *gcP**, mmlDisplayPixmap** *destP**, int32** xc**, int32** yc**, f24Dot8** xscale**, f24Dot8** yscale**, int32** angle**, int32*** pPtsLst)

Draws a closed set of lines, each beginning where the last ends, and with a line from the last point to the first point. The pointer *ptlist* points to this many points.

Parameters such as the line type, thickness of the lines, color, z comparison mode (if applicable), alpha value (if applicable), z value (if applicable), etc., are taken from the graphics context *gcP*.

The parameters are:

int32 *xc* **—** The x-coordinate of the center of the polygon. This only uses the lower 16 bits of *int32*. It accepts negative values.

int32 *yc* **—** The y-coordinate of the center of the polygon. This only uses the lower 16 bits of *int32*. It accepts negative values.

f24Dot8 *xscale* **—** The x scaling factor of the polygon. This is in 8.8 format (i.e., the maximum value is *0x7FFF)*.

f24Dot8 *yscale* **—** The y scaling factor of the polygon. This is in 8.8 format (i.e., the maximum value is *0x7FFF)*.

int32 *angle* **—** This is the number of clockwise rotations (full or fractional) in 16.16 format. It accepts negative values.

int32* *pPtsLst* **—** This is a pointer to an array of points. Each point contains the y-coordinate in the upper 16 bits and the x-coordinate in the lower 16 bits. For example the point 0x000f000a would represent a point with an x-coordinate of 15 and a y-coordinate of 10. Each point-coordinate is added to the corresponding center-coordinate (*xc* or *yc*); and multiplied by the corresponding scaling factor (xscale or yscale).

The following escape codes can be freely intermingled with the list of points to either change the line characteristics or break the polygon continuity. Please note that all references to fields from the **m2dLineStyle** structure refer to *gcP->defaultLS.*

Escape Code	Description
0x80000000	Change the line width of the next line segments using the value of the next entry in the list. This escape code has the side effect of generating a break in the polyline for all line types (**eLine1** to **eLine6**, inclusive).
0x80000001	Terminate the polyline. This should always be the last entry in the list of points.
0x80000002	Do not connect the previous point to the next point.

Escape Code	Description
0x80000003	Set *foreColor* using the value of the next entry in the list. *NOTE*: If the current line type is set to **eLine6**, then the value in *colorBlend1* also has to be set. (See section 3.2.2) For example, if the new color value is 0xA28E2C00 and *colorBlend1* is 0x3F, then the value *foreColor* is 0xA28E2C3F. For **eLine3clut** and **eLine7clut**, the color index should be in the format 0x*nn*00, effectively leaving the first byte empty. For example, a color index of 0x81 should be entered as *0x8100*.
0x80000004	Set *foreColor2* using the value of the next entry in the list, and set *foreColor* using the value after the next entry in the list (after the value used by *foreColor2*). *NOTE*: If the current line type is set to **eLine6**, then the value in *colorBlend1* also has to be set. (See section 3.2.2) For example, if the new color value is 0xA28E2C00 and *colorBlend1* is 0x3F, then the *foreColor* is 0xA28E2C3F. The same applies to *foreColor2*. For **eLine3clut** and **eLine7clut**, the color index should be in the format 0x*nn*00, effectively leaving the first byte empty. For example, a color index of 0x81 should be entered as *0x8100*. This applies to *foreColor* and *foreColor2*. The indices between *foreColor* and *foreColor2* will be used as pixel values, between the 2 endpoints. In other words, *foreColor* and *foreColor2* will be used as the minimum and maximum of a range of color indices. The value of *foreColor* can be less than or greater than *foreColor2*.
0x80000005	Change the alpha value using the value of the next entry in the list. This escape code has the side effect of generating a break in the polyline for all line types (eLine1 to eLine6, inclusive). However, the alpha value will only be used by **eLine3**, **eLine5** and **eLine6**.

In order to draw a closed polygon, the first polygon point should also be the last polygon point in the list. The following array is an example of a closed polygon in a C-language program:

```
int LineList[] =
{
      0xff9cff9c,
      0xff9c0064,
      0x00640064,
      0x0064ff9c,
      0xff9cff9c,
      0x80000001
};
```

9. Ellipse Draw Functions

9.1.1 m2dInitEllipseStyle (not supported in SDK > 0.87)

void m2dInitEllipseStyle(**mmlGC*** gcP**, m2dEllipseStyle*** ellipseS**,**
f16Dot16 ratio**, mmlColor** color**,**
f24Dot8 xScale**, f24Dot8** yScale**,**
int alpha**, int32** fill)

Call to initialize an **m2dEllipseStyle** object with a specific border *width*, *color*, *x/y eccentricity*, *alpha*, and *fill* value. The **m2dEllipseStyle** is described in detail in section 3.2.3.

9.1.2 m2dDrawEllipse (not supported in SDK > 0.87)

void m2dDrawEllipse (**mmlGC** **gc*, **mmlDisplayPixmap** **V*,
int32 *xCenter*, **int32** *yCenter*, **int32** *rad*)

Draws an ellipse with center at *xCenter*, *yCenter* and a radius of *rad*. The craphics context *gc* provides scale factor that combine with *rad* to provide a circle or an ellipse. The thickness of the circle, its color, etc. are also provided by the graphics context *gc->defaultES*.

Initially, the graphics context has default scale values of 1.0 so that the function draws circles with the radius *rad*.

9.1.3 m2dDrawStyledEllipse(not supported in SDK > 0.87)

void m2dDrawEllipse (**mmlGC** **gc*, **mmlDisplayPixmap** **V*,
m2dEllipseStyle **stylePtr*,
int32 *xCenter*, **int32** *yCenter*, **int32** *rad*)

Draws an ellipse with center at *xCenter*, *yCenter* and a radius of *rad*. The eccentricity factors, color, etc. are taken from the **m2dEllipseStyle**. These properties override the properties in *gc->defaultES*, but do not replace them.

9.1.4 m2dDrawQuadArc (not supported in SDK > 0.87)

void m2dDrawQuadArc (**mmlGC** *gcP, **mmlDisplayPixmap** *destP,
int32 *xCenter*, **int32** *yCenter*,
int32 rad, **int32** quadrant)

Draws a 90-degree arc in a quadrant with center at *xCenter*, *yCenter* and a radius of *rad*. Quadrants are numbered 1 to 4 in a clockwise direction starting with the bottom right which is numbered 1 and ending with the top right which is 4.

The thickness of the arc, its color, etc. are provided by *gc->defaultES*.

9.1.5 m2dDrawStyledQuadArc(not supported in SDK > 0.87)

void m2dDrawStyledQuadArc (**mmlGC** *gcP, **mmlDisplayPixmap** *destP,
m2dEllipseStyle **stylePtr*,
int32 *xCenter*, **int32** *yCenter*,
int32 rad, **int32** quadrant)

Draws a 90-degree arc in a quadrant with center at *xCenter*, *yCenter* and a radius of *rad*. Quadrants are numbered 1 to 4 in a clockwise direction starting with the bottom right which is numbered 1 and ending with the top right which is 4.

The eccentricity factors, color, etc. are taken from the **m2dEllipseStyle**. These properties override the properties in *gc->defaultES*.

10. Box Functions

10.1.1 m2dInitBox

mmlStatus m2dInitBox(**mmlGC** ******gcP*, **mmlDisplayPixmap** ******destP*,
m2dBox* *bP*,
int *maxWidth*, **int** *maxHeight*, **int** *maxLineWidth*)

Allocates memory for a box object whose maximum dimensions are given by the parameters. When the box object is no longer needed, the memory should be released with ***m2dReleaseBox***. Returns **eOK**, unless memory could not be allocated, in which case it returns **eSysMemAllocFail**.

10.1.2 m2dDrawBox

void m2dDrawBox(**mmlGC** ******gcP*, **mmlDisplayPixmap** ******destP*,
m2dBox* *bP*, **int** *width*, **int** *height*,
int *lineWidth*, **int** *left*, **int** *top*, **mmlColor** *color*)

Draw a box in the specified pixmap. All of the properties of the box are provided as parameters; *height*, *width*, *linewidth*, and *color*. The position of the top left corner is also passed as an argument. This is the top left corner of the outside edge of the box. If the box is already visible on the screen, this function erases the current outline and draws one with the new arguments.

10.1.3 m2dEraseBox

void m2dEraseBox(**mmlGC** ******gcP*, **mmlDisplayPixmap** ******destP*, **m2dBox*** *bP*)

Erases the outline that currently represents the box. The same box may be later redrawn at the same position with ***m2dRedrawBox***.

10.1.4 m2dRedrawBox

void m2dRedrawBox(**mmlGC** ******gcP*, **mmlDisplayPixmap** ******destP*, **m2dBox*** *bP*)

Redraw a box that has been previously hidden with ***m2dEraseBox***. It is drawn at the same position and with the same properties as when it was erased.

10.1.5 m2dReleaseBox

void m2dReleaseBox(m2dBox* *bP*)

Release the memory allocated for this box object. The box may not be used again until it is reinitialized with ***m2dInitBox***.

11. Arrow Functions

11.1.1 m2dInitArrow

mmlStatus m2dInitArrow(mmlSysResources* *srP*, **m2dArrow*** *aP*,
uint32 *wide*, **uint32** *high* **)**

Initializes an arrow to have the size *wide* * *high*. There are no restrictions on the dimensions of an arrow. However, SDRAM is generally scarce, so arrows should be kept small, and the return status should always be checked to see if memory was successfully allocated.

Returns **eOK**, unless memory could not be allocated for the arrow. An allocation failure is reported as **eMerMemAllocFail**.

11.1.2 m2dSetArrowPixel

void m2dSetArrowPixel(mmlGC* *gcP*, **m2dArrow*** *aP*,
int *x*, **int** *y*, **mmlColor** *color* **)**

Sets the pixel at coordinates *x,y* to *color*. X ranges from 0 to wide-1. Y ranges from 0 to high-1. Coordinate 0,0 is at the top left of the arrow rectangle. The alpha part of color is respected. A value of 00 in the least significant byte makes the pixel completely opaque. A value of 0xFF makes the pixel completely transparent. Intermediate values cause blending with the background pixel.

11.1.3 m2dShowArrow

void m2dShowArrow(mmlGC* *gcP*, **m2dArrow*** *aP*,
mmlDisplayPixmap* *destP*, **coord** *left*, **coord** *top* **)**

Causes an arrow to be displayed at coordinates left and top in an existing **mmlDisplayPixmap**. The part of the pixmap that is covered up is saved so it can be restored by **m2dHideArrow** or **m2dMoveArrow**.

11.1.4 m2dHideArrow

void m2dHideArrow(mmlGC* *gcP*, **m2dArrow*** *aP* **)**

Hides an existing arrow, by restoring the previously saved image of the screen under the arrow. Any time an application intends to draw on the pixmap in an area

overlapping the arrow, the application should first do **m2dHideArrow** and then **m2dRedrawArrow** after updating the pixmap.

11.1.5 m2dRedrawArrow

void m2dRedrawArrow(mmlGC* *gcP*, **m2dArrow*** *aP* **)**

Redraws a previously hidden arrow at the same position it occupied when it was hidden.

11.1.6 m2dMoveArrow

void m2dMoveArrow(mmlGC* *gcP*, **m2dArrow*** ***aP*, mmlDisplayPixmap*** ***destP*, coord *newLeft*, coord *newTop*)**

Move an existing arrow from its current location to a new location, possibly in a new pixmap. The screen under the current location of the arrow is restored to its state prior to displaying the arrow.

11.1.7 m2dDeleteArrow

void m2dDeleteArrow(mmlSysResources* ***srP*, m2dArrow*** ***aP*)**

Deletes an arrow object. The video memory in SDRAM used for the arrow image is released, as is the memory used to save and restore the screen beneath the arrow.

12. Text and Fonts

The Merlin Media Library makes it easy to display richly styled text in any of the world's languages.

The basic element of the text API is a *textLine*. The client can specify the style properties of any substring of the line of text.

Style properties include typeface, point size, emphasis, weight, text color, anti-aliasing, translucency, and other optional properties that depend on the chosen typeface.

Style properties may have enumerated values such as 'emphasis' = **eNormal** or **eItalic**; 'weight' = **eRegular** or **eBold**; or they may have numeric values such as 'pointsize' = 12, 13, …; or 'translucency' = 0.0, .. 0.5, ..1.0.

The font rendering engine is the T2K type engine from Type Solutions Incorporated. This engine will render TrueType fonts and fonts in the proprietary T2K format. Thousands of typefaces can be used in the Merlin Media Library.

A sans serif typeface, specifically intended for use on low resolution displays, is provided as part of the text system. The name of this font is SysFont. A bold weight is also provided with the name SysFontBold.

A client can add other fonts by licensing TrueType fonts and adding them to the registered font list. To save space, TrueType fonts can be converted to the T2K format by VM Labs or TypeSolutions Inc.

The following functions are provided for manipulating fonts and drawing text:

12.1.1 mmlGetRegisteredFonts

void mmlGetRegisteredFonts(mmlFontContext* *fP*, **mmlFont** *fonts*[ ], **int ****numFonts* **)**

Call with ***numFonts** = 0, to learn the number of currently registered typefaces. *numFonts* will return the number of registered typefaces. Call with ***numFonts** = N, to obtain an array of the first N registered typefaces.

An **mmlFont** is a font reference structure that is passed to other font manipulation functions.

12.1.2 mmlAddFont

mmlFont* mmlAddFont(**mmlFontContext** *fc*, **textCode** *typefaceName*[ ],
typeTechnology *tech*, **uint8*** *fontLocation*, **int** *size* **)**

Call to register a new typeface. Specify the memory location of the font and it's type technology. Returns an **mmlFont** reference. For example, to use the system font, make the call mmlAddFont(fc, "foo", eT2K, SysFont, SysFontEnd-SysFont);

12.1.3 mmlRemoveFont

void mmlRemoveFont(mmlFontContext *fcP*, **mmlFont** *font* **);**

Call to remove a registered typeface. Free all the memory associated with this font.

12.1.4 mmlGetFontName

void mmlGetFontName(mmlFont f, textCode ** **nameP);**

Return a string containing the name of a registered font.

12.1.5 mmlSetTextProperties

void mmlSetTextProperties (**mmlFontContext** *fP* , **mmlFont** *fontP*,
int *pointsize,* **mmlColor** *foreColor*,
mmlColor *backColor,* **textMix** *copyMode,*
int *copyFlags,* **f16Dot16** *angle*)

Set the properties of the default text style in the font context.

The *pointsize* parameter is the vertical height (in pixels) of a line of text including the leading between adjacent lines.

The *foreColor* parameter is the color of the text.

The *backColor* parameter is used to color all the non-text pixels in the line if the copy mode is e**SourceCopy**.

The *copyFlags* parameter controls more properties of the text draw. A value of **kFillRect** causes the entire rectangle to be filled. A value of zero causes the drawing to stop after the last letter in the string.

The *angle* argument is expressed in fractional clockwise rotations, but is currently ignored.

The *copyMode* argument can have the values **eOpaque**, **eBlend**, or **eClutAlpha**. In **eOpaque** mode, a box of **backColor** is drawn with text in **foreColor** anti-aliased against the **backColor** box. In **eBlend** mode, the existing framebuffer will be read, and text in the **foreColor** will be anti-aliased against the existing pixels.

The **eClutAlpha** mode can only be used for text drawn in a framebuffer that uses the **eClut8** pixel format, and also will be drawn in the OSD plane. In this mode, the text pixel values are divided by the graphics context parameter *gc.textDiv*, and added to *gc.TextBase* to produce an index into the current Color LookUp Table that has been set in the NUON chip. (See bios setClut function). The produced index is clipped to the range *gc.TextMin* and *gc.TextMax*. The CLUT values used for text should be arranged so that opacity *decreases* from *gc.TextMin* to *gc.TextMax*.

Note: The **eClutAlpha** mode expects the palette entries to decrease in opacity as the index value increases. This is the same convention as is used in CLUT based line and ellipse drawing. This is the reverse of the previously documented **eAlpha** mode which is no longer supported.

The text pixel values are actually mask values ranging from 0 to 63 that represent coverage of a pixel. These values are used for blending and anti-aliasing of text over video or against a background.

Two important text properties must be set in the graphics context. The field **gc.translucentText** will ordinarily be set to 0, but if the **copyMode** parameter is **eOpaque** and the text is to be displayed on top of video using translucent foreground or background colors, then **gc.translucentText** should be set to 1.

The **gc.textWidthScale** value is a **16Dot16** fixed point number representing the ratio of the width of text characters to the height. A value of 0x10000 corresponds to 1.0 (the default value) and is generally the correct setting. Other values can be used to produce condensed or expanded text. In particular, a value of 0x8000 (0.5) should be used if horizontal pixel doubling is being used in the graphics plane.

12.1.6 mmlInitScaledTextStyle

void mmlInitScaledTextStyle (**mmlTextStyle*** *tsP* **, mmlFont** *fontP***,**
int *pointsize,* **mmlColor** *foreColor***,**
mmlColor *backColor,* **textMix** *copyMode,*
int *copyFlags,* **f16Dot16** *angle,*
f16Dot16 *xScale*)

Set the properties of a text style.

The *pointsize* parameter is the vertical height (in pixels) of a line of text including the leading between adjacent lines.

The *foreColor* parameter is the color of the text.

The *backColor* parameter is used to color all the non-text pixels in the line if the copy mode is eS**ourceCopy**.

The *copyMode* parameter specifies how to combine the text with the background. The only value supported currently is 0 which is **eSourceCopy**.

The *copyFlags* parameter controls more properties of the text draw. A value of **kFillRect** causes the entire rectangle to be filled. A value of zero causes the drawing to stop after the last letter in the string.

The *angle* argument is expressed in fractional clockwise rotations, but is currently ignored.

The *xScale* argument is a fractional scale value that can be used to compress or expand the width of characters. The default value is 0x10000 (1.0).

12.1.7 mmlInitTextStyle (deprecated)

void mmlInitTextStyle (**mmlTextStyle*** *tsP* **, mmlFont** *fontP*,
int *pointsize,* **mmlColor** *foreColor*,
mmlColor *backColor,* **textMix** *copyMode,*
int *copyFlags,* **f16Dot16** *angle*)

Set the properties of a text style. Same as **mmlInitScaledTextStyle()** except that the scaling option is always set to 1.0.

12.1.8 mmlSetTextStyle

void mmlSetTextStyle (**mmlFontContext** *fc*, **mmlTextStyle*** *tsP* **)**

Set the properties of the default text style in the font context from an existing **mmlTextStyle**. The default **mmlTextStyle** in the **mmlFontContext** always controls any text rendering.

12.1.9 mmlSetTextModel

void mmlSetTextModel(mmlFontContext fcP, textModel model);

Set the current text model to eNewModel or eOldModel (default). This call should be made immediately after the mmlInitFontContext call. In the old text model, the size of text is automatically adjusted so that a string will always fit in a box that is

pointSize high, even if the string contains the letter with the highest ascender and the letter with the lowest descender.

In the new text model, no automatic size adjustment is made. So text will be rendered at exactly the pointSize specified in the style or text properties. This will generally be larger than in the old text model. If lines are drawn so that their baselines are exactly pointSize above each other, the letters in the bottom line may collide with the letters in the upper line. This will be particularly noticeable if the eOpaque copy mode is used rather than the eBlend mode.

A safe line height can be found by making the mmlGetLayoutMetrics or mmlGetLayoutStyleMetrics call and setting the line height to the sum of the font ascent and font descent.

12.1.10 mmlSimpleDrawText

void mmlSimpleDrawText(**mmlFontContext** *fP* **, mmlDisplayPixmap** *V***,**
textCode *txt[ ],* **int** *numLetters***, m2dRect** *rP*)

Draw *numLetters* of text in the pixmap beginning at the top left corner of the rectangle. Clip the text to the rectangle boundaries. Use the typeface and style characteristics specified in the **mmlFontContext**. The **m2dRect** bottom right coordinates are modified to describe the rectangle actually occupied by the rendered text.

12.1.11 mmlSimpleDrawBaseline

void mmlSimpleDrawBaseline(mmlFontContext *fcP***, mmlDisplayPixmap*** *screenP***, textCode** *str[],* **int** *numLetters***, int** *baseX***, int** *baseY***);**

Draw *numLetters* of text in the pixmap beginning at the point (baseX, baseY). This point is taken to be the text baseline point where drawing of the string will begin. The only clipping done is to the displayPixmap itself. Use the typeface and style characteristics specified in the **mmlFontContext**.

12.1.12 mmlGetTextBox

void mmlGetTextBox(**mmlFontContext** *fP***, textCode** *txt[ ],*
int *first,* **int** *last,* **m2dRect*** *rP* **)**

Returns the coordinates of the rectangle that will be required to contain the substring of the text beginning at character *first* and ending with character *last* when it is rendered as a single line of text using the typeface and style characteristics specified in the **mmlFontContext**. The initial values of the **m2dRect** are the left top position

where the complete text string is to be drawn and the right bottom coordinates of the clipping rectangle that must contain the text.

12.1.13 mmlGetStyleLayoutMetrics

mmlLayoutMetrics* mmlGetStyleLayoutMetrics (**mmlFontContext** *fP*,
mmlTextStyle* *tsP*)

Returns a pointer to an **mmlLayoutMetrics** object. This object contains info useful for laying out text in the specified **mmlTextStyle**. See section 3.2.9 for information about the format of this object.

12.1.14 mmlGetLayoutMetrics

mmlLayoutMetrics* mmlGetLayoutMetrics (**mmlFontContext** *fP*)

Returns a pointer to an **mmlLayoutMetrics** object. This function is similar to the **mmlGetStyleLayoutMetrics** function, except that it returns metrics for the current style in the font context (i.e. the last style to be created by **mmlSetTextProperties** or **mmlSetTextStyle**).

12.1.15 mmlCharKindQ

charKind mmlCharKindQ(textCode *c*, **textEncoding** *standard*)

Returns the kind of character represented by the **textCode** *c* in the specified standard. Returns **eLetter**, **eNumber**, **eWhiteSpace**, **ePunctuation**, or **eExtra**.

This function can be used to do line breaking at ends of words. The only encoding standard supported in this version is **eAscii**. Future versions will support **eUnicode**.

13. Sequence Functions

13.1.1 mmlOpenSeq

mmlStatus mmlOpenSeq(mmlGC* *gcP*, **mmlSequence*** *seqP*, **int** *numCmds* **);**

Opens an **mmlSequence** object to record up to *numCmds* operations. After calling ***mmlOpenSeq***, further drawing commands are not executed; instead, they are collected as a sequence for later execution. To stop recording call ***mmlCloseSeq***. This will cause drawing commands to resume being directly executed.

Returns **eOK**, unless memory could not be allocated for *numCmds* operations. An allocation failure is reported as **eSysMemAllocFail**.

13.1.2 mmlCloseSeq

void mmlCloseSeq(mmlGC* *gcP*, **mmlSequence*** *seqP* **)**

Stops recording drawing operations into the currently open sequence object. The sequence can then be executed with ***mmlExecuteSeq***. or more commands can be called directly. The sequence can be executed any number of times until it has been released with ***mmlReleaseSeq***.

13.1.3 mmlExecuteSeq

void m2dExecuteSeq(mmlGC* ***gcP***, *gcP*, **mmlSequence*** *seqP* **)**

Execute the commands previously recorded into the sequence object. If a recorded command uses arguments obtained from the graphics context, the values that were in the graphics context at the time of recording are used, not the values present at the time of execution.

13.1.4 mmlReopenSeq

mmlStatus mmlReopenSeq(mmlGC* ***gcP***, **mmlSequence*** *seqP*,
int *numMoreCmds* **)**

Reopens a previously recorded sequence to append more commands. If memory can not be allocated for *numMoreCmds*, **eSysMemAllocFail** is returned, else the function returns eOK. *NumMoreCmds* can be 0. When commands are being recorded and allocated memory has been used, an attempt will be made to allocate more memory. However, if this attempt fails, there is no direct method of reporting the failure; the

command is simply not recorded. Programs should either insure that there is sufficient memory by using ***mmlOpenSeq*** and ***mmlReopenSeq*** for known numbers of commands, or the program should monitor the sequence object's field value *seqP->numCommands* to verify that it has been incremented after each command has been recorded.

13.1.5 mmlReleaseSeq

void mmlReleaseSeq (mmlGC* *gcP*, mmlSequence* *seqP*)

Releases the memory previously allocated for the sequence. The sequence can not be used again until it is again initialized with ***mmlOpenSeq***. ***mmlReleaseSeq*** can be called on a currently open sequence, however, this would not generally be done, because the sequence could not be executed.