VM LABS

De Re

The NUON Application Programming Guide

Revision 0.04
20-Feb-01

Note: This document is continually updated to reflect the current state of the NUON development system hardware and software. If you have a version that is more than five or six months old, it is likely out of date.

Please address comments or report errors to Mike Fulton at VM Labs (EMAIL: mfulton@vmlabs.com).

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

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

Table of Contents

1. INTRODUCTION 1-1

1.1 Welcome To NUON 1-1

1.1.1 Who is this document for? 1-1

1.1.1.1 No Experience Required? 1-1

1.1.2 Where else to look? 1-1

1.2 What Can NUON Do? 1-2

1.2.1 Video 1-2

1.2.2 Audio 1-3

1.2.3 2D Graphics 1-3

1.2.4 3D Graphics 1-3

1.2.5 Controllers 1-3

1.2.6 Media Access 1-3

2. VIDEO 2-1

2.1 Pixel Types 2-1

2.1.1 Pixel Color Information 2-1

2.1.1.1 CLUT-based Pixels 2-2

2.1.1.2 True Color / High Color Pixels 2-2

2.1.1.3 MPEG Pixels 2-2

2.1.2 Alpha Channel Value 2-2

2.1.3 Z-Buffer Value 2-3

2.1.3.1 Avoiding Z-Buffer Errors 2-3

2.1.3.2 NUON Z-Buffer Sizes 2-4

2.2 Display = Main Channel + Overlay Channel 2-4

2.2.1 Alpha Channel 2-5

2.2.1.1 How Alpha Blending Works 2-5

2.2.1.2 Palette Based Pixel Modes 2-6

2.2.1.3 Alpha Channel – Not Just For Hardware 2-6

2.2.1.4 Pixel Formats 2-6
2.2.1.5 Overlay Channel Uses 2-6
2.2.1.6 Overlay Channel Drawbacks 2-6
2.3 Frame Buffer Creation 2-7
2.3.1 Double & Triple Buffering 2-7
2.3.1.1 Double-buffering 2-7
2.3.1.2 Triple-buffering 2-8
2.3.1.3 NUON Hardware Buffer Modes 2-9
2.3.2 Frame Buffer Initialization With MML2D 2-9
2.3.2.1 Buffer Allocation 2-10
2.3.2.2 Hardware Double-buffering or Triple-buffering 2-10
2.3.3 Frame Buffer Initialization With M3DL 2-11
2.3.4 Frame Buffer Initialization With mGL 2-11
2.4 Video Hardware Setup 2-11
2.4.1 BIOS Functions For Video Initialization 2-11
2.4.2 MML2D Functions For Video Initialization 2-12
2.4.3 M3DL Functions For Video Initialization 2-12
2.4.4 mGL Functions For Video Initialization 2-12

3. AUDIO 3-1
3.1 LIBSYNTH –vs– LIBNISE 3-1
3.1.1 Machine Resources 3-1
3.1.2 Deciding which library to use 3-2
3.2 Audio Initialization 3-3
3.3 Dolby Pro-Logic Encoding 3-3

4. 2D GRAPHICS 4-1
4.1 M3DL Library 4-1
4.2 MML2D Library 4-1
4.3 Impulse Graphics Library 4-1
4.4 DMA Blitting 4-1

5. 3D GRAPHICS 5-1

5.1 M3DL Library 5-1

5.2 mGL Library 5-1

5.3 MML3D Library 5-1

5.4 Caviar Library 5-1

6. MEDIA ACCESS 6-1

6.1 Overview of Media Access Functions 6-1

6.2 Making a NUON Data File 6-1

6.2.1 Data File Strategy 6-2

6.2.2 Data File Concatenation 6-2

6.3 NUON Data Tool 6-2

6.3.1 How Does It Work 6-2

6.3.1.1 Adding Files To The List 6-3

6.3.1.2 Changing The List Order 6-3

6.3.1.3 Refreshing The List 6-3

6.3.1.4 Options 6-3

6.3.1.5 Creating Your Data File 6-4

6.3.1.6 Include File Generation 6-4

6.3.1.7 The Home Directory 6-5

6.3.1.8 Global Options 6-6

6.3.1.9 Using The .H Include File Definitions 6-6

6.4 Making a NUON DVD 6-7

6.4.1 Program Authentication 6-7

6.4.2 Disk Layout 6-8

6.4.2.1 NUON Folder 6-8

6.4.2.2 VIDEO_TS & AUDIO_TS 6-8

This page intentionally left blank.

1. Introduction

1.1 Welcome To NUON

Congratulations on becoming a NUON programmer. This document will give an experienced programmer the information needed to start from scratch and create a fully operational NUON application.

At the end of this document, we'll actually create a simple application. It will be simple in most respects, but it will demonstrate the same structure and methodology of a more complex application.

1.1.1 Who is this document for?

This document is aimed at experienced game programmers who are working on their first NUON project. However, even if you're working on a non-game project, this document may still be helpful.

1.1.1.1 No Experience Required?

If you're not an experienced programmer, but you've been assigned to work on a NUON project, then it's very likely that somebody has made a big mistake. While it's not technically impossible, an embedded system like NUON is a poor choice of platform to use while you're still learning to program.

If you are an experienced programmer, but have never worked on a game project before, then it would be a good idea to read a book or two on game programming before you get too far into your NUON game project. At the very least, you need to familiarize yourself with some of the terminology used in the field.

See the chapter on ***Recommended Reading*** for some recommended titles.

An embedded system like NUON is not the ideal platform to learn game programming, but it can be done.

1.1.2 Where else to look?

This document is a tutorial, not a reference. Please make sure that you have familiarized yourself with ***The Hitchhiker's Guide to NUON*** and the other documentation provided in the NUON SDK. It's not necessary for you to read

everything from cover to cover, but you should at least have an idea of where to look for additional documentation when you have a need for it.

1.2 What Can NUON Do?

The NUON processor is designed with the following ideals in mind:

- An instruction set optimized for the needs of media processing, yet flexible enough for general purpose use.
- Providing multiple, symmetrical processors that can be used for a variety of purposes.
- Avoid having dedicated-purpose co-processors that waste space and power when they aren't being used.

1.2.1 Video

The video subsystem of NUON has the following features:

- 720 x 480 output resolution (NTSC or PAL60)
  720 x 576 output resolution (PAL50)
- Can use arbitrary size bitmap for frame buffer, with hardware scaling to output resolution
- 2-tap or 4-tap horizontal filter
- 2-tap or 4-tap vertical filter.
- Separate main channel and overlay channel
- Hardware alpha-channel blending for overlay channel
- 4-bit, 8-bit, 16-bit, or 32-bit per pixel display modes.
- Special MPEG pixel type for direct display of MPEG macroblock data.
- Hardware 16-bit or 32-bit Z buffer, integrated with DMA controller
- All 8mb of SDRAM can be used for display frame buffers
- Textures for 3D graphics or 2D sprites can be located anywhere in memory.

1.2.2 Audio

The audio subsystem and libraries for NUON have the following features:

- 32 voice output
- Audio streaming from disc
- Variable panning
- Reverb
- General MIDI-compatible wavetable synthesizer

1.2.3 2D Graphics

There are numerous ways to create 2D graphics on NUON.

1.2.4 3D Graphics

There are numerous ways to create 3D graphics on NUON.

1.2.5 Controllers

The NUON BIOS provides a flexible and yet expandable method of dealing with game controllers.

The first design goal for NUON's controller handling was to give applications the chance to use controller types that did not yet exist when the application was written.

With most consoles, the system assigns an arbitrary ID number to each controller type. So if a game knows that controller type $21 has certain capabilities, it can act accordingly. But if controller type $33 comes along, it might not be recognized. Then the controller would have no idea if a certain button was never being pressed because that was the user's choice, or if because that button did not exist on controller type $33.

Instead of assigning static ID numbers, the NUON BIOS allows an application to determine what capabilities are available.

1.2.6 Media Access

More to come…

This page intentionally left blank

2. Video

Regardless of the application, there are two primary aspects of video initialization that must be considered:

- Frame Buffer Creation
- Video Hardware Setup

We'll be discussing frame buffers in detail later on, but first let's go over the basic video capabilities of NUON.

2.1 Pixel Types

NUON pixels contain up to three different types of information.

- Pixel color
- Alpha Channel
- Z-Buffer

2.1.1 Pixel Color Information

The pixel color information is always present in some form. There are a wide variety of formats used by NUON, but they break down into the following three categories:

- Color Look-Up Table (CLUT) pixels
- True Color / High Color pixels
- MPEG pixels

It's important to note that we are talking about pixel types that are supported by the NUON hardware, and more specifically, by the video display generator.

Keep in mind that software routines can use just about any pixel type they want. Some formats may not always be an efficient choice, but that decision is up to the programmers who write the graphics code.

2.1.1.1 CLUT-based Pixels

NUON supports pixel modes where the pixel value represents an index into an array of color definitions. This array is known as either a *CLUT* (for Color Look-Up Table) or *palette.*

NUON supports either 4-bit or 8-bit CLUT pixel types, which allow either 16 or 256 different colors.

Because of the limited number of colors available, a CLUT-based pixel mode is not usually the first choice for the overall display. And in fact, NUON does not allow the use of these modes for the display buffer assigned to the main video channel. They can be used for the display of the overlay video channel and in other situations. (More information on the main channel and overlay channel is coming in an upcoming section.)

2.1.1.2 True Color / High Color Pixels

The terms *true color* and *high color* are used to indicate a pixel mode where the pixel value itself directly represents the color value that will be displayed. The difference between the terms is simply that *true color* normally refers to pixels with at least 24 bits of color value, while *high color* is used to refer to pixels with 12, 15, or 16 bits of color value.

Pixel modes supported by NUON include several variations that provide 16 bits of color information, and several others that provide 24-bits.

2.1.1.3 MPEG Pixels

NUON also understands a special pixel format that is designed specially for the display of decoded MPEG video data. This format is not ordinarily used by applications.

2.1.2 Alpha Channel Value

The *alpha channel* value is an extra piece of information that is stored either with each pixel, or with each entry in a color palette.

Note that the alpha channel is an optional component of a pixel.

We'll discuss the alpha channel in more detail in section 2.2.

2.1.3 Z-Buffer Value

One of the biggest problems in 3D graphics is how to determine which objects are hidden by other objects, or how one object intersects another. This has to be resolved so that everything is drawn with the right priority relative to everything else. The most popular solution to this problem is the Z-buffer. Let's briefly explain how a Z-buffer works:

As a 3D object (e.g. a polygon) is rendered, the coordinates of the object are processed and ultimately there is a Z-depth value is associated with each pixel. This represents the distance of that pixel from the viewing plane.

When it comes time to store each pixel into the frame buffer, we need to check if it is in front of what's already in the frame buffer at that location. This is determined by reading the Z-depth value that is already contained in the frame buffer and comparing it with the Z-depth value for the pixel that was just computed.

If the new Z-depth value is closer to the viewing plane, then the new pixel information is written to the frame buffer, including the Z-depth value. Otherwise, the new pixel information is discarded and the old information is maintained.

There are drawbacks to using a Z-buffer. First, it takes more memory since the frame buffer has to contain Z-depth values in addition to color values. Also, it takes extra memory access to read the existing pixel values before writing each pixel.

However, the drawbacks are strongly outweighed by the advantages on the NUON system. For one thing, the Z-buffer mechanism provides a method for multiple processors to work on rendering 3D graphics without worrying about objects being drawn out of the proper sequence. This allows us to take advantage of the parallel processing power that NUON provides.

Secondly, the NUON DMA system handles Z-buffer comparisons automatically, and has several options for different types of comparisons. This means that using a Z-buffer is very easy and largely automatic.

2.1.3.1 Avoiding Z-Buffer Errors

When an application attempts to create a 3D world, one of the important considerations is the accuracy and precision of the numbers used to store and calculate object positions.

The size of the Z-buffer values determines the accuracy. If your Z-buffer resolution was only 4-bit, then there would only be 16 possible Z-depth values. It would be very easy to have errors in your rendering as a result. For example, the Z-depth value of one object might be represented as 7 when it's really supposed to

be 7.9. Another object might also be 7 when it's really supposed to be 7.1. Because the precision is too low, these two objects may not be assigned the proper priority relative to each other. Priority would depend on the order in which the objects are drawn, which may not provide the desired results.

It's easy to see that 4-bits is not enough resolution, and fortunately we don't have to worry about it because NUON supports Z-buffer values of either 16 or 32 bits, depending on the pixel type selected. However, it is important to note that there will be situations in which 16 bits is still not enough precision.

If you have a scene in your world with objects that are very close and objects that are very far away, you will be stretching the bounds of your Z-buffer precision. In a situation like this, 16 bits of Z-depth may not be enough.

The best thing is to simply keep track of situations where the Z-buffer precision may be stretched. If you don't see any *significant* rendering errors, then don't change anything.

2.1.3.2 NUON Z-Buffer Sizes

On NUON, the Z-buffer is optional, but when you use it, the resolution is tied to the size of the rest of the pixel. The 16-bit pixel modes can have 16-bits of Z-depth, and 32-bit pixel modes get 32 bits.

The MPEG, 4-bit and 8-bit pixel modes cannot use the Z-buffer.

2.2 Display = Main Channel + Overlay Channel

Most devices like video game consoles, computer video cards, and DVD players generate a video display by using a block of computer memory to hold values that correspond to different brightness and color values. To create the display, the video hardware reads those values in the right sequence and uses them to modulate a video signal that is being created. The *frame buffer* is the area of memory that contains the information used by the video hardware to create the display.[1]

The NUON system has the ability to read information from two separate frame buffers and blend the information together to create a display. Each of these separate frame buffers is known as a *channel*.

On NUON, the *main* channel is normally used to contain the background of your display. For most applications, this is the only channel used.

---

[1] In a more general sense, the term *frame buffer* refers to an area of memory used by a program to store or create an image, even if that image is not currently being displayed.

On NUON, the *overlay* channel is optional. When active, it is blended with the contents of the main channel by the NUON video hardware.

2.2.1 Alpha Channel

To combine the overlay and main channels together, the NUON's video display generator must have some means of deciding how much information to take from each channel. This is done using a device known as the *alpha channel,* which is an extra value stored along with each pixel. This value is used to specify the degree of transparency for each pixel in the overlay buffer. This process is called *alpha blending*.

2.2.1.1 How Alpha Blending Works

The exact process used for alpha blending on NUON depends on the video mode used by the overlay channel.

When using a display mode with 32 bits per pixel, each pixel contains 8 bits of alpha channel information. This provides 256 possible levels of transparency which can be mixed in any combination.

Using a display mode with 4 or 8 bits per pixel is very similar, except that the alpha channel information is not maintained for each individual pixel. Instead, each 32-bit palette entry contains 8 bits of alpha channel information. This provides 256 possible levels of transparency overall, but each palette entry may control any number of pixels.

If the display mode uses 16 bits per pixel, then the NUON video hardware handles alpha blending a bit differently, because there is no alpha channel information for each pixel, and there is no color look-up table.

The alpha channel value may be represented by a variable number of bits. For a display mode using 32-bit pixels, the alpha channel value is 8-bits. This means there are 256 possible levels of transparency. But with a 16-bit pixel mode, the alpha channel is just 1 bit. So there are only two possible levels, either completely transparent or completely opaque.

We'll consider that an alpha channel value with no bits set corresponds to a value of 0.0, and that an alpha channel value with all of the bits set corresponds to a value of 1.0.

A value of 0.0 (no alpha channel bits set) indicates that there should be no transparency at all, so the video display hardware outputs whatever color was indicated by the overlay channel pixel. A value of 1.0 (meaning all alpha channel

bits are set) indicates that the overlay is 100% transparent, so the video display hardware outputs the color indicated

Based on the alpha channel value, a pixel from the overlay may be completely invisible, completely opaque, or anywhere between.

When the overlay channel is active, the alpha channel value associated with each pixel is used to determine the relative opacity of that pixel.

2.2.1.2 Palette Based Pixel Modes

For palette-based pixel modes, the alpha channel information is not stored with each individual pixel of the frame buffer. Instead, it is stored with the palette information. This palette information is 32-bits per entry, which includes 8-bits of alpha channel information.

2.2.1.3 Alpha Channel – Not Just For Hardware

As far as the NUON video display generator is concerned, the alpha channel is only considered important when the overlay channel is active.

However, it is important to note that the alpha channel may also be used by graphics rendering libraries to indicate things like transparency of individual objects. Keep that in mind, but remember that for right now we're talking about how the hardware works.

2.2.1.4 Pixel Formats

The two video channels do not have to be the same format. The main channel may be showing a movie using MPEG-format pixels at 720x480 resolution while the overlay channel uses a 4-bit pixel mode at 360x240 to display graphics.

2.2.1.5 Overlay Channel Uses

The overlay channel may be used for many purposes. For example, a game might use it to contain the game score and other such information. Since these items don't always change rapidly, using the overlay channel means that you've reduced the amount of work required to render the rest of the display.

2.2.1.6 Overlay Channel Drawbacks

Using the overlay channel may not always be desirable, however. The main drawbacks are:

- More complicated video setup
- Uses more memory
- Overlay channel cannot scale frame buffer with as much flexibility as the main channel.
- Uses more bandwidth on the main bus, since the video hardware must read two separate frame buffers and merge them together.

2.3 Frame Buffer Creation

Creating a frame buffer is largely a matter of allocating a block of memory in SDRAM and initializing whatever structure is being used to deal with it.

There are functions for frame buffer creation in several NUON libraries. First we'll discuss a couple of the concepts that are important to managing the frame buffer, then we'll discuss examples using each library in turn.

2.3.1 Double & Triple Buffering

The techniques of double-buffering and triple buffering are important tools for smooth graphics programming. NUON has special support for these techniques that may be used in certain cases.

2.3.1.1 Double-buffering

Double buffering is a technique used to avoid certain types of visual glitches during screen updates.

Two frame buffers are used. At any one time, the video display hardware uses one for the current display while the other one is used to render a new image to be displayed.

When the rendering process is finished, a "page flip" operation is performed to direct the video display hardware to the new buffer. This is synchronized to the video hardware's vertical blanking period so that the video hardware never makes the switch in the middle of drawing the display. Otherwise we would get the glitch that we are trying to avoid.

After the "page flip" has taken place, then the video hardware switches to the newly rendered image. Then the program can continue rendering the next frame in what is now the off-screen buffer. The result is a smooth and instant transition from one image to the next.

2.3.1.2 Triple-buffering

Triple buffering is similar to double buffering, except there is secondary purpose in mind aside from avoiding screen update glitches. By using three buffers, the program is able to refresh the display at a more consistent interval.

With normal double-buffering, the application spends a certain amount of time waiting for a vertical blank period so that it can perform the page-flip. This works out OK when the time required for rendering does not vary a lot. You waste some processing time, but the display refresh rate remains consistent.

However, suppose that the time required for rendering each frame is not so consistent. Let's say it takes 1.8 video fields to render a frame, and 2.1 video fields to render the next one. If we are synchronizing to a display with a 60hz refresh rate, the first frame's rendering time[2] will round up to 2.0 fields, which gives us a frame rate of 30 frames per second. However, the second frame's time will round up to 3.0 fields, for a frame rate of 20 frames per second. That means this program's screen refresh rate will be bouncing back and forth between 20 and 30 frames a second. This is likely to look jittery to the user.

When we are double-buffering and have to wait for a vertical blank period, we cannot do anything to either image buffer. Otherwise we would corrupt either the display that's just been rendered, or the image that is currently being shown by the video hardware. We have to wait until after the page flip has completed before we can do any additional rendering.

If we use triple-buffering, however, the program does not usually have to just wait when it finishes rendering a frame. It now has an extra buffer and can begin rendering a new frame right away, even though the frame it just finished may not actually be shown by the video hardware yet. The only time it has to wait is when it has two completed buffers that haven't been displayed yet.

Since we're not waiting around for vertical blank to happen, we have to handle the page flip differently. The easiest method is to install an interrupt handler that will be called during the video hardware's vertical blank period. The page flip is then done inside the interrupt handler, while the non-interrupt code is working on rendering the next frame.

This means that if we take 1.8 video fields to render a frame, and 2.1 video fields to do the next, the overall time is still just 3.9 fields for those two frames. That rounds up to 2.0 fields per frame, for a frame rate of 30 frames per second.

---

[2] For the purpose of discussion, we include all the processing that an application does between one frame and the next under the general heading of "rendering time" even though parts of that processing may have nothing to do with graphics.

It's still possible for the frame rate to vary, but it should happen less often. The difference is sometimes considerable. The result is a significant overall improvement in the smoothness of your display.

2.3.1.3 NUON Hardware Buffer Modes

The NUON system supports two different methods of double or triple buffering. First, it supports the traditional model where you have two or three complete and separate buffers, each of which can be located anywhere in SDRAM.

However, when you are using a Z-buffer for your graphics rendering, this can take a lot of memory because each buffer is independent of the others. With this in mind, the NUON hardware provides a special method for double-buffering or triple buffering that can be used for the **e655Z** 16-bit pixel mode.[3]

In these modes, a single Z-buffer is shared with either 2 or 3 rendering/display buffers, rather than using a separate Z buffer for each frame buffer.

Some of the NUON libraries select this mode automatically when your application requests the right combination of pixel type and buffer count.

See your *MMP-L3B Programmer's Guide* for more information about these modes.

2.3.2 Frame Buffer Initialization With MML2D

The function ***mmlInitDisplayPixmaps()*** is used to initialize one or more **mmlDisplayPixmap** structures. The code below will show how to initialize two screen buffers that will be used for a double-buffered display.

```
mmlSysResources  gl_sysRes;
mmlDisplayPixmap gl_screenbuffers[2];
int              gl_displaybuffer, gl_drawbuffer;

void init_screenbuffers()
{
        // Make sure gl_sysRes stuff is setup
        mmlPowerUpGraphics( &gl_sysRes );

        // Initialize index values for gl_screenbuffers[] array
        gl_displaybuffer = 0;
        gl_drawbuffer = 1;

        // Create & clear each buffer

        mmlInitDisplayPixmaps(   &gl_screenbuffers[gl_displaybuffer],
                                 &gl_sysRes,
```

[3] The **e655Z** mode provides 16-bits of YCrCb color data and 16-bits of Z buffer for each pixel.

```
                                SCREENWIDTH, SCREENHEIGHT,
                                e888Alpha, 1, NULL );

        mmlInitDisplayPixmaps(  &gl_screenbuffers[gl_drawbuffer],
                                &gl_sysRes,
                                SCREENWIDTH, SCREENHEIGHT,
                                e888Alpha, 1, NULL );
}
```

First we declare the global variables used to track the library and system resources and our frame buffers. First is the *gl_sysRes* variable. This is a structure used to store context information for several of the NUON graphics libraries.

Inside the ***init_screenbuffers()*** function, we start out by initializing the *gl_sysRes* structure with the ***mmlPowerUpGraphics()*** function. This is typically done as one of the very first few steps of an application, so the ***init_screenbuffers()*** function is intended to be called either at or near the top of the application's ***main()*** function.

Next we initialize the two variables that will be used to indicate which buffer is being used for rendering and which one is used for the display.

Finally, we have two calls to the ***mmlInitDisplayPixmaps()*** function. This function will initialize the fields of the **mmlDisplayPixmap** structure and calculate the correct set of flags for NUON DMA operations. In this example, we're calling for the **e888Alpha** 32-bit pixel mode.

2.3.2.1 Buffer Allocation

The last parameter to the ***mmlInitDisplayPixmaps()*** function is the address in SDRAM where the frame buffer(s) being initialized will be located. If you pass a NULL pointer, then the library will automatically calculate how much memory is required for your request and allocate that amount.

2.3.2.2 Hardware Double-buffering or Triple-buffering

The MML2D library supports the special double-buffer and triple buffer 16-bit video modes described. If you select **e655Z** as your pixel type and request that either 2 or 3 buffers should be created, then it will automatically use the shared buffer mode.

You may override this by allocating your separate buffers individually as shown in this example, instead of all at once.

2.3.3 Frame Buffer Initialization With M3DL

Instead of using a single general-purpose function for initializing frame buffers with any arbitrary combination of attributes, the M3DL library includes several different specialized functions. Each function handles a single combination of attributes.

Part of the reason it works this way is that the M3DL library gives your application the option of rendering into a buffer that uses either a generic RGB format or the NUON's native YCrCb-format. Certain features like transparency blending and colored lighting effects do not work properly unless you use RGB mode, but the current generation of NUON hardware does not support RGB mode. Therefore, when the M3DL library renders an RGB-based frame buffer, it must be converted to YCrCb before it can be displayed. Fortunately, the M3DL library handles this process for you, and does so quite efficiently.

For our sample, we will be using just one frame buffer format. For additional information on the other options available, please see the chapter titled ***Frame Buffer Setup*** in the M3DL Library documentation.

2.3.4 Frame Buffer Initialization With mGL

The mGL library

2.4 Video Hardware Setup

Video hardware setup on NUON is very simple, because the NUON BIOS takes care of the low-level details. This is very important, because the NUON chip may be used with a wide variety of video display hardware depending on the type of device and the manufacturer.

2.4.1 BIOS Functions For Video Initialization

The most commonly used BIOS function for video configuration is this:

```
_VidSetup(  void *framebuf,
            int dmaFlags,
            int framebuf_width,
            int framebuf_height,
            int filtertype );
```

This function presumes that you want just the main video channel, no overlay. The five arguments to this function describe the frame buffer format.

If more control over the video is desired, such as enabling the overlay channel, then the following function can be used:

```
_VidConfig( VidDisplay *disp,
            VidChannel *main,
            VidChannel *overlay,
            void *reserved );
```

The ***disp*** structure describes the overall video configuration for things like vertical and horizontal filtering, the display position and size, border color, and so forth.

The ***main*** and ***overlay*** structures describe the frame buffers that are assigned to those video channels. To disable either channel, you can pass through a NULL pointer.

For more information, please see the BIOS documentation.

2.4.2 MML2D Functions For Video Initialization

The MML2D library has a simple function for video display initialization:

```
mmlSimpleVideoSetup( mmlDisplayPixmap *framebuf,
                     mmlSysResources *sysRes,
                     int filtertype );
```

This function takes just three arguments: a pointer to a structure describing a frame buffer, a pointer to the MML library system resource information, and a value that indicates what type of video filter should be used.

The **mmlDisplayPixmap** structure used here would be one initialized by the function **mmlInitDisplayPixmap()**, as described in section 2.3.2.

For more information, please see the MML2D Library documentation.

2.4.3 M3DL Functions For Video Initialization

The M3DL library does not provide any functions for video display initialization. Please use the BIOS and/or MML2D functions.

2.4.4 mGL Functions For Video Initialization

The M3DL library does not provide any functions for video display initialization. Please use the BIOS and/or MML2D functions.

3. Audio

There are two primary libraries for audio on NUON. Both libraries provide a variety of features, but they differ in some significant respects.

The LIBNISE library is oriented around playback of PCM sample data and streaming audio. The LIBSYNTH library includes some basic PCM playback features, but is oriented towards music synthesis and MIDI playback.

First we'll compare the features of the two libraries, then we'll step through the requirements of initialization and how to use the library.

Please note that neither of these libraries provide

3.1 LIBSYNTH –vs– LIBNISE

The table below shows which features are available in each library.

Feature	Available in LIBNISE?	Available in LIBSYNTH?
Memory-Resident PCM Sample Playback	Yes	Yes
Wavetable Synthesis	No	Yes
MIDI Playback	No	Yes
Streamed Playback of audio from DVD or other media	Yes	No
Dolby Pro-Logic encoding of PCM data	Yes	Yes
Multi-tap Reverb	No	Yes
Simple Echo	Yes	No
3D Panning	Yes	Yes

The libraries are mutually exclusive. That is, you can only use one or the other in your application. The main reason for this is that for the features that both libraries have in common, the API is the same. This is done in order to keep things simple and easy to use.

3.1.1 Machine Resources

The LIBNISE library is very compact and has a minimal impact on the system. The interrupt code required for PCM playback and for managing audio data streaming from DVD is designed to co-exist with the NUON mini-BIOS. This

means that using LIBNISE does not require a dedicated processor. However, it does mean that the mini-BIOS must be active when using LIBNISE.

The LIBSYNTH library is significantly larger than LIBNISE in terms of memory usage and has a larger impact on the system. First of all, it requires a dedicated processor. Secondly, the current version of LIBSYNTH cannot co-exist on the same processor with the NUON mini-BIOS. However, the LIBSYNTH library does not depend on the mini-BIOS at all, which means it's possible to shut down the mini-BIOS while using LIBSYNTH.

The main reason LIBSYNTH uses a lot of memory is that it requires a memory-resident wavetable containing sample data for the 128 different voice patches that are part of the General MIDI standard. This requires about 1.5mb of RAM. Aside from this, a large RAM buffer is also required for the LIBSYNTH reverb feature. LIBNISE substitutes a simple echo effect that requires less processing and less memory.

The table below compares the machine resource requirements of both libraries.

Feature	LIBNISE	LIBSYNTH?
Requires dedicated MPE?	No	Yes
Can co-exist with mini-BIOS?	Yes	Yes, but not on same MPE
Requires mini-BIOS?	Yes	No
Memory Footprint	about 15kb	1.8mb

3.1.2 Deciding which library to use

The decision of which audio library you should use in your application usually comes down to this question: Do you want your game music to use streaming audio or MIDI?

This is not an easy question to answer, so we'll try provide the arguments for either side so that you can make your own decision.

Streaming audio is a nice, easy way to provide background music in your game, with minimal impact to the rest of the system. And a DVD disc can easily contain a couple of hours of different music selections to choose from.

Streaming audio also gives you absolute control over what your music will sound like. With MIDI, there are always differences in the sound between one MIDI device and another. Furthermore, some musical effects are either difficult or impossible to accomplish via MIDI.

On the other hand, streaming audio is potentially less interactive than using the synthesizer. Don't forget that the synthesizer does not limit you to simple MIDI playback. Your program can take direct control of the synth and generate music dynamically. This allows you to continually customize the music to match the gameplay, something that streaming audio can't do.

Using the synth also means that you can use the full bandwidth of your media for reading other data. You can load a new level of your game while music is playing. It is possible to mix streaming audio with reading other data, but this is difficult. For one thing, you have to purposely limit how much you read at once or else you'll cause your streaming audio to fail. This slows down your data reading. Also, it increases the chances that a disc error will occur and cause a streaming audio failure.

3.2 Audio Initialization

Regardless of which library you use, your program uses one of two functions to initialize everything:

AUDIOInit()	Allow the library to allocate memory used for buffers and the MPE used for processing.
AUDIOInitX()	Pass a pointer to an AUDIO_RESOURCES structure that contains information about which MPE to use, buffer addresses, etc.

3.3 Dolby Pro-Logic Encoding

The NUON audio libraries feature real-time Dolby Pro-Logic surround-sound encoding that enables your games to use surround sound when connected to an audio receiver capable of decoding such a signal.

Because the NUON processor is capable of decoding Dolby Digital 5.1 audio streams, many NUON players include six channel analog outputs. However, this depends on the individual model and manufacturer, so some models may include only a stereo analog output.

When your NUON player is connected to an audio receiver

The NUON audio libraries allow you to pan sounds from front to rear as well as from right to left.

The libraries create four discrete channels of audio which are sent to your NUON player's analog audio outputs.[4]

four channels (front left, front right, rear left, rear right) of audio created by the libraries are automatically encoded into a Dolby Pro-Logic signal. This encoded signal is then output to the stereo audio outputs. If your NUON player is connected to an audio receiver that includes Dolby Pro-Logic decoding, then a surround-sound signal can be generated.

[4] NUON is capable of decoding Dolby Digital 5.1 audio streams, and many NUON players will include six channel analog outputs. However, this depends on the individual model and manufacturer. Some models may include only a stereo analog output, along with a digital coaxial or optical output.

4. 2D Graphics

There are numerous ways to create 2D graphics on NUON. So let's start out by categorizing the various options.

4.1 M3DL Library

4.2 MML2D Library

4.3 Impulse Graphics Library

4.4 DMA Blitting

5. 3D Graphics

5.1 M3DL Library

5.2 mGL Library

5.3 MML3D Library

5.4 Caviar Library

6. Media Access

The media access libraries for NUON are intended to offer the best possible mix of efficiency, ease of use, flexibility, and device abstraction. They were designed especially to avoid certain problems which have been seen on other console systems.

6.1 Overview of Media Access Functions

The Media Access functions are documented in the NUON BIOS documentation. The three most commonly used functions are:

Function	Description
_MediaOpen()	Opens a file on a particular file media device for reading. On some devices, the desired file is specified by the filename. On others, there is only one file per device.
_MediaClose()	Closes a previously opened file and frees the device handle for use with another file.
_MediaRead()	Reads data sectors from the specified device. This function only reads complete sectors, but the sector size may vary between devices. This function is normally asynchronous, meaning that it returns immediately before the data has actually been transferred from the device. To determine when the data read has completed, or to detect errors, etc., the program specifies a callback routine.

6.2 Making a NUON Data File

The NUON media access libraries are designed with the idea that an application will place all of its data into one or more large data files, rather than a large number of smaller files.

The main reason for this is the desire to have maximum efficiency at runtime. By using a single large data file instead of many small ones, we shift some of the task of file management from runtime to development time and also eliminate certain hardware-based delays that are introduced when you have a large number of files to deal with.

6.2.1 Data File Strategy

Ideally, a NUON application would have just a single data file named NUON.DAT. However, this isn't always practical because the UDF file system used by DVD has a maximum file size of 1 gigabyte, which is only a fraction of the total storage capacity of the format. Therefore, an application may have as many data files as needed.

However, even though there is no restriction on having multiple data files, it's still most efficient to have as few as possible. The reasons for this include:

- The BIOS has a limited number of file access handles available.
- Each time you open a file, the system must read the disk directory.
- Unlike computer systems where there may be a multi-megabyte disk cache, the NUON is a closed system where memory is limited and there is only a very small cache available.

6.2.2 Data File Concatenation

In order to accommodate the idea of using a small number of large data files, an application developer must have a means to combine smaller data files together into a single large file.

This may be done in a variety of ways. We suggest using a tool provided for this express purpose, the ***NUON Data Tool***.

6.3 NUON Data Tool

The ***NUON Data Tool*** is a software application that runs under Microsoft Windows. It may be used to combine a list of smaller data files together into a single large data file.

The program maintains a list of all the files that will be combined to create the target NUON data file. You may add files to this list, delete files from the list, change the position of items in the list, or change the options for each file.

Once you have all of the desired files added to your list, then you can tell the program to create the NUON data file.

6.3.1 How Does It Work

Upon launch, the program normally starts with an empty list window. Please note that you can also launch the program by double-clicking a list file, in the same way you might launch Microsoft Word by double-clicking on a .DOC file.

You may also select ***New*** from the *File* menu to open a new, empty list window.

6.3.1.1 Adding Files To The List

Once you have an empty list window, the easiest way to add files is to simply drag them from Windows Explorer into the ***NUON Data Tool*** window and release them. This adds the files to the list.

You can also add files by right-clicking the mouse over the list window. When the context popup menu appears, the first item should be ***Add New File***. This will bring up a file selector dialog so that you may select a file to be added to the list.

Once files have been added to the list, you can add more items, delete items, alter the order, or change the alignment options for individual items.

6.3.1.2 Changing The List Order

You can change the order of items in the list by right-clicking the mouse over an item. This will bring up the context popup menu, which includes choices for moving files up or down in the list by an item at a time, or by all the way to the top or bottom.

6.3.1.3 Refreshing The List

If any of the files in the list are altered after the list has been created, you should refresh the list to correct it. This will cause the program to read the current file size and time/date stamp information.

You may refresh an individual file entry by right-clicking the mouse over an item. This will bring up the context popup menu, which includes a choice for ***Refresh Size & Time/Date Stamp***. If the file cannot be found, you will be asked if the entry should be removed from the list.

You may also refresh all the files in the list by selecting the menu item titled ***Refresh Size & Time/Date Stamp*** in the *Options* menu. If any items cannot be found, the file size will be reset to zero.

6.3.1.4 Options

For each file you add to the list, the ***NUON Data Tool*** gives you the option of:

- Output C/C++ Header Files — This option will cause the ***NUON Data Tool*** to create include files for C & C++ that contain preprocessor definitions describing each file in your list.

6.3.1.5 Creating Your Data File

Also see ***Include File Generation*** below.

6.3.1.6 Include File Generation

When the ***NUON Data Tool*** creates your data file, your application needs a method to determine where each piece of data is located and how big it is.

To accommodate this, the ***Create NUON Data File*** dialog includes an option for creating a special C/C++ include file with definitions for the location and size of each piece of data. There is also an option for creating the equivalent Llama assembly language file.

For each file in your list, the program creates six definitions. Each definition begins with a modified version of the original source file's complete pathname with all of the special filename characters like ":" or "\" converted to underscore ("_") characters. After the pathname comes a special suffix that indicates the type of information represented by the definition. For example, if you had a source file named:

```
C:\NUONSpaceCannon\Data\Level1\background.jpg
```

Then for the C/C++ include file option, you would see a set of six definitions that look like this.

```
#define C__NUONSpaceCannon_Level1_background_jpg_FILEREF     (2)
#define C__NUONSpaceCannon_Level1_background_jpg_BLOCK       (12)
#define C__NUONSpaceCannon_Level1_background_jpg_NUMBLOCKS   (210)
#define C__NUONSpaceCannon_Level1_background_jpg_NUMBYTES    (429602)
#define C__NUONSpaceCannon_Level1_background_jpg_BLOCKOFFSET (0)
#define C__NUONSpaceCannon_Level1_background_jpg_BYTEOFFSET  (24576)
```

The meaning of each of the suffixes is defined in the table below:

Suffix	Meaning
_FILEREF	Indicates which entry this was in the list. Useful mainly for reference purposes.
_BLOCK	Indicates the starting sector number for the data, relative to the beginning of the file. This the value that would be passed to the ***_MediaRead*** function.
_NUMBLOCKS	Indicates how many blocks you must read in order to obtain all of the data for the item. Note that an item may not be aligned to the beginning of a sector, so this value may be larger than you would expect from the file size
_NUMBYTES	Indicates the size in bytes of the original data file

Suffix	Meaning
_BLOCKOFFSET	Indicates the offset in bytes where the data begins, relative to the first data block of the item. For example, if the _BLOCK definition says 12, and _BLOCKOFFSET says 356, then the data item begins at byte offset 356 of sector 12 of the data file generated by the ***NUON Data Tool***. This will always be zero for files where you have specified sector boundary alignment.
_BYTEOFFSET	Indicates the offset in bytes where the data begins, relative to the overall data file generated by the ***NUON Data Tool***. Useful mainly for reference purposes.

6.3.1.7 The Home Directory

The *Home Directory* is simply the directory that contains all of the source files used to generate your data file. For example, suppose that your project has the following three source files:

```
C:\NUONSpaceCannon\Level1\background.jpg
C:\NUONSpaceCannon\Level2\background.jpg
C:\NUONSpaceCannon\Level3\background.jpg
```

Each of the files is located in its own folder, but all of those folders are contained in:

```
C:\NUONSpaceCannon\
```

So this would be your *Home Directory*. Note that in order to really take advantage of the *Home Directory* feature, all of the source data files must be located in the same folder of the same drive.

As we saw earlier, the complete pathname of each item is used to create the definitions that describe the data file being generated. However, as our example in section 6.3.1.6 indicates, this can result in some very long definitions.

When you specify the *Home Directory*, the ***NUON Data Tool*** will automatically remove that part of the pathname from the beginning before generating definitions. So if our home directory was specified as shown above, then our definitions for the first file would be:

```
#define Level1_background_jpg_FILEREF      (2)
#define Level1_background_jpg_BLOCK        (12)
#define Level1_background_jpg_NUMBLOCKS    (210)
#define Level1_background_jpg_NUMBYTES     (429602)
#define Level1_background_jpg_BLOCKOFFSET  (0)
#define Level1_background_jpg_BYTEOFFSET   (24576)
```

instead of:

```
#define C__NUONSpaceCannon_Level1_background_jpg_FILEREF     (2)
#define C__NUONSpaceCannon_Level1_background_jpg_BLOCK       (12)
#define C__NUONSpaceCannon_Level1_background_jpg_NUMBLOCKS   (210)
#define C__NUONSpaceCannon_Level1_background_jpg_NUMBYTES    (429602)
#define C__NUONSpaceCannon_Level1_background_jpg_BLOCKOFFSET (0)
#define C__NUONSpaceCannon_Level1_background_jpg_BYTEOFFSET  (24576)
```

As you can see, this will eliminate a lot of typing in the long run.

6.3.1.8 Global Options

The ***Global Options*** menu item in the *Options* menu allows you to specify several default options that will apply to your file list.

- Default sector alignment for each file added to the list
- Sector size of the target media. This is almost always 2048 bytes for DVD or CDROM, but could also be different values for other types of media. This value is only used to determine sector alignment.
- Home Directory
- Filename for Data File Generation

6.3.1.9 Using The .H Include File Definitions

Using the definitions provided in the generated include file is quite simple. If you are trying to access the data for **background.jpg** and you have definitions that look like this:

```
#define Level1_background_jpg_FILEREF      (2)
#define Level1_background_jpg_BLOCK        (12)
#define Level1_background_jpg_NUMBLOCKS    (210)
#define Level1_background_jpg_NUMBYTES     (429602)
#define Level1_background_jpg_BLOCKOFFSET  (0)
#define Level1_background_jpg_BYTEOFFSET   (24576)
```

Then your call to read the data will look something like this:

```
error = MediaRead( media_handle, MCB_END,
                   Level1_background_jpg_BLOCK,
                   Level1_background_jpg_NUMBLOCKS,
                   buffer, callbackfunction );
```

The *media_handle* parameter is the successful return value from a previous call to *MediaOpen(). The MCB_END* parameter indicates that the callback will occur after the last block is read.

The next two parameters are taken from the definitions generated by the ***NUON Data Tool*** and they represent the starting sector of the data, relative to the

beginning of the open file, followed by the number of blocks that must be read 9in order to get all of the data.

Next we have the memory address of the buffer that will receive the data.

Last, we have a function pointer to our callback routine. Because we're using ***MCB_END***, this function will be called when the data has been completely read into memory, or if an error occurs.

See the NUON ***BIOS Documentation*** for more information on the details of the media access functions.

6.4 Making a NUON DVD

A NUON DVD is, in most respects, a standard DVD-ROM disc which contains a NUON directory that contains the program and data files required for the application.

Optionally, the disc may also contain the files for a DVD video disc such as might be found on a standard movie disc. This would allow the disc to show a video when the disc is used on a non-NUON system.

Finally, the disc may also optionally contain files intended for a PC

6.4.1 Program Authentication

A program that is intended to run on a standard consumer NUON player must go through a process called *authentication***.** This is a process by which the application's executable program file is specially encoded in a particular way.

At runtime, when a NUON player runs a program, it tests the authentication information to make sure that it is genuine and properly licensed. If it passes the test, then the program is loaded and executed.

Consumer NUON players are not capable of running a program unless it has gone through the authentication process. If the file is not properly coded, it will be rejected as being a non-executable file.

The authentication process is something that nobody besides VM Labs can do. It is one of the methods by which VM Labs enforces the requirement that all NUON software must be properly licensed and tested for hardware compatibility.

When an application is finished and ready for production, the developer must submit the program files to VM Labs so that they may go through the authentication process before the final disc is mastered.

Authentication is not required for programs running on a NUON development system.

6.4.2 Disk Layout

In most cases, all of the files for a NUON application will be located in the NUON folder located in the root directory of the disc. Other information may also be located on the disc.

6.4.2.1 NUON Folder

This folder should contain the following folders:

- NUON.N16 — This is a raw 16-bit YCrCb image file, 352 pixels wide by 288 pixels tall. On NTSC systems, the top 24 lines and bottom 24 lines are discarded. On PAL systems, all 288 lines are shown.

  This image will be displayed while your main program file is loading. This file is normally supplied by VM Labs and will contain license and copyright information.

- NUON.RUN — This is the application's main executable program file that is automatically loaded at boot time. For consumer NUON machines, this file must be authenticated.

  Note that this program may be something small and simple that plays a movie or shows a splash screen, and then launches the real main program.

- NUON.DAT — The name "NUON.DAT" is the default choice of name for your program's first data file. There is no actual requirement that the filename be "NUON.DAT", but it is common practice.

6.4.2.2 VIDEO_TS & AUDIO_TS

Optionally, a NUON application disc may also have a VIDEO_TS folder and an AUDIO_TS folder in the root directory. These contain the files required for any DVD-Video or DVD-Audio material that may be placed on the disc to be used when the disc is inserted into a non-NUON player.

The content of these directories can be authored using industry-standard authoring tools.