VM LABS

De Re NUON

The NUON Application Programming Guide

Revision 0.08
30-Oct-01

This document is not quite finished, but there's plenty of useful information already and we wanted to give you the opportunity to see it now. We'll be making updates as time and resources permit.

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

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

Table of Contents

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.

This document also contains a number of useful bits of information that might be useful later in your project.

1.1.1.1 No Experience Required?

If you're not an experienced programmer at all, and 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 far from the ideal platform for somebody who is learning to program for the very first time. This is true for a wide range of reasons, including:

- The documentation is not written for beginners. It presumes you already have a certain level of expertise with similar tools and programming libraries.
- The programming tools are generally very different from what's available for systems like Microsoft Windows or Macintosh, and may not be documented with a beginner in mind.
- The system has many eccentricities compared to a regular computer.
- Technical details are protected by non-disclosure agreements, so there are few options for assistance.

If you are an experienced programmer, but have never worked on a game project before, then you're somewhat better off. An embedded system like NUON is not the ideal platform to learn the specifics of game programming, but it can be done. You should read some books on game programming before you get too far into your NUON game project. It's OK if those books discuss using a different platform and graphics library, like Direct3D under Microsoft Windows. While the details of the programming libraries will be different, the basic concepts involved will be similar in many ways. At the very least, such books will allow you to familiarize yourself with the terminology.

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

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.

1.1.3 A Note About The NUON Open Platform

This document was originally created to meet the needs of commercial developers working on bringing NUON compatible products to market. It therefore discusses the entire range of tools and programming libraries that VM Labs makes available to commercial developers. However, because of licensing concerns, documentation requirements, hardware requirements, and other such issues, the NUON Open Platform Software Development Kit does not provide all of the same tools and programming libraries.

Because of the nature of this document, it would be very difficult to maintain if there were separate versions for commercial developers and for NUON Open Platform. Therefore, we ask that users of the NUON Open Platform please keep in mind that there may be items discussed within this document to which you do not have access. We will attempt to make a note of such items where applicable.

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, (mostly) 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, except that image width must be a certain multiple depending on the pixel format.
- Features hardware scaling to scale source frame buffer up or down to output resolution
- 2-tap or 4-tap horizontal video filter
- 2-tap or 4-tap vertical video 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. (4-bit & 8-bit available only on overlay channel.)
- Special MPEG pixel type for direct display of MPEG macroblock data.
- Hardware 16-bit or 32-bit Z buffer, integrated with DMA controller
- Any portion of SDRAM (8mb) can be used for display frame buffers
- Textures for 3D graphics or 2D sprites can be located anywhere in memory.

Some features may not be supported by all programming libraries.

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
- Dolby Digital AC-3 decoding

Audio features largely depend more on the individual library than the NUON hardware. The NUON audio system is ultimately a relatively simple six-channel audio output system driven by a powerful DSP (the NUON processor) that performs audio decoding, music, synthesis, voice mixing, etc.

This means that the audio features are subject to change depending on what a programmer decides to implement in the NUON audio libraries. For example, the 32-voice output listed above is a relatively arbitrary limit that could be either higher or lower.

1.2.3 2D Graphics

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

- MML2D graphics library
- M3DL graphics library (provides basic 2D sprite features in addition to 3D graphics features)
- Impulse graphics library
- Render it yourself . You can directly manipulate raster-format images and then copy them to the screen, or you can perform DMA at the pixel level on a frame buffer or off-screen buffer.

We'll learn more about these various options throughout this document.

1.2.4 3D Graphics

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

- M3DL Graphics library (provides 2d sprites, 3d graphics rendering, & geometry transformation)
- mGL graphics library (similar to OpenGL)
- MML3D graphics library. This library is generally considered to be a basic shell or prototype for creating your own custom 3D graphics library, because it does not have all of the features usually considered necessary for a game.
- Create your own 3D graphics library

We'll learn more about these various options throughout this document.

1.2.5 Controllers

The NUON system adopts a straight-forward yet flexible method of accessing a wide variety of game controllers ranging from joysticks to steering wheels to fishing reels.

1.2.6 Memory Cards

NUON can use three types of memory card device:[1]

- SmartMedia cards — This uses a special SmartMedia card adapter, or a game controller or other peripheral that includes a SmartMedia card slot. These are the same SmartMedia cards used by devices such as digital cameras and personal music players (MP3) and they are not NUON-specific. Sizes from 2mb to 128mb are supported.
- Embedded Memory Cards — This is a memory card device that's embedded into the NUON console itself, or in a NUON peripheral device such as a game controller or multi-controller adapter. Sizes from 2mb to 128mb are supported.

[1] At the time of this writing, memory card libraries and hardware have not yet been finalized, and no release date is available.

- NUON Memory Cards — This is a proprietary memory card device which connects to a NUON peripheral such as a game controller or multi-controller adapter. Sizes from 2mb to 128mb are supported.

All of the above types of memory card are identical as far as the NUON memory card libraries are concerned. Only the physical packaging is different.

1.2.6.1 Custom Memory Card Manager Application

VM Labs plans to eventually supply a memory card manager application that can be customized with application-specific artwork. This application will be able to copy files, move files, delete files, format a card, and perform other file management tasks.

1.2.6.2 Accessing Digital Camera & MP3 Player Files

When you're using SmartMedia cards, the memory card libraries support the ability to access files created by digital cameras, MP3 players, and so forth. However, what you actually do with those files is up to you to figure out.

1.2.7 Media Access

The NUON BIOS provides a way to access data from DVD discs quickly and efficiently. Accessing data from disc is done asynchronously, so that your application can perform other processing while waiting for data reads to complete.

Access to CDROM (including CD-R and CD-RW media) is not supported at the time of this writing, but it is expected that such support will be made available as either a BIOS update or linkable library within the next several months.

Please note that CD-R and CD-RW discs are not compatible with the DVD drive mechanism used by certain older models of NUON DVD players. In particular, the Samsung N2000 and Toshiba SD-2300 do not support CD-R and CD-RW discs.

For more information about media access, please see the NUON BIOS Documentation.

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. Just keep in mind that custom formats will have to be converted before the frame buffer may be used as a display.

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.[2]

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.

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 determining the relative priority of the information from each channel. Otherwise, you would only see the overlay channel whenever it was used.

To blend the channels together, NUON uses a mechanism known as the *alpha channel*, which is simply an extra value that gets 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 first factor in the process is the pixel's color value. If the color value is zero, then that pixel is always treated as 100% transparent. If the color value is non-zero, then the alpha channel value is used to determine the degree of transparency.

Alpha channel values range from 0 to 255. An alpha channel value of 0 corresponds to 0% transparent, meaning that none of the main channel information will be displayed. An alpha channel value of 255 corresponds to about 99.6% transparent. Intermediate alpha channel values provide different degrees of transparency. Note that there are actually 257 levels of transparency available, since a pixel color value of 0 is used for 100% transparency.

The alpha channel value comes from different places depending on the pixel format used by the overlay channel:

* 32-bit pixels have 24-bits of Y-Cr-Cb color information and 8-bits of alpha channel.
* 16-bit pixels have 16-bits of packed Y-Cr-Cb color information, and no alpha channel bits at all. Therefore, you can have either 100% transparency or 0% transparency, based on the color value being zero or not.
* 4-bit pixels have a color palette with 16 entries that are 32-bits each. The 4-bit pixel value is converted to whatever 32-bit value is contained in the corresponding palette entry.
* 8-bit pixels have a color palette with 256 entries that are 32-bits each. The 8-bit pixel value is converted to whatever 32-bit value is contained in the corresponding palette entry.

[2] 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.

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

The color palette used for 4-bit and 8-bit pixel formats can be set using the BIOS function ***_VidSetCLUTRange***().

2.2.1.2 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. Most typically, the alpha channel may be used in a texture that is applied to a 3D model to indicate transparent portions of the texture.

Keep that in mind, but remember that for right now we're talking about how the hardware works.

2.2.1.3 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.

2.2.1.4 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.

2.2.1.5 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. Note that it is possible to specify a combination of main channel resolution and pixel format and overlay channel pixel and format that cannot be properly displayed.

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 a 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 from one frame to the next. You may 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[3] 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, and this does not happen very often.

Since we’re not just 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.

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.[4]

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.

[3] 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.

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

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,
                                 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, including **MML2D**.

Next we have an array named *gl_screenbuffers*, which is an **mmlDisplayPixmap** structure. This structure maintains the information required to describe a bitmap image that is suitable for use by the display hardware.

Finally, the variables *gl_displaybuffer* and *gl_drawbuffer* are used as indices into the *gl_screenbuffers* array to indicate which buffer is currently used for the display and which is currently being used for drawing the next frame.

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 variables *gl_displaybuffer* and *gl_drawbuffer*.

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.

If you do not pass a NULL pointer, note that the library expects that you're handling all of the proper memory allocation requirements.

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 for the display. 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

Standard MML2D functions are used to initialize the desired frame buffers and then information about these buffers is passed to mGL. The initial video configuration is done by calling either MML2D or the BIOS video setup functions. After this, mGL functions are used to handle buffer switching. Please see the various mGL library sample programs for more information.

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. After initializing the frame buffer, you must grab the required values out of the M3DL drawing context structure and pass them to the BIOS video setup functions. For additional information, please see the chapter titled ***Frame Buffer Setup*** in the M3DL Library documentation.

2.4.4 mGL Functions For Video Initialization

The initial video configuration is done by calling either MML2D or the BIOS video setup functions. After this, mGL functions are used to handle buffer switching. Please see the various mGL library sample programs for more information.

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.

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

The audio libraries create four discrete channels of audio that are encoded into a Dolby Pro-Logic signal. This is a stereo matrix-encoded signal that gets sent to your NUON player's stereo analog audio outputs.

Although many NUON players do provide Dolby Digital 5.1 audio decoding during DVD movie playback, the audio libraries do not provide runtime Dolby Digital encoding. This would require a substantial cost in terms of system resources that would not be acceptable in virtually all cases.

If no runtime Dolby Digital encoding is available, then why not output discrete channels via the six channel analog outputs? Simple, not all players have them.[5]

Plus, we don't really have six channels of data. We have front left & right, and rear left and right. There is no center channel or low frequency subwoofer channel. Those channels could be created, but it would require a significant amount of additional processing which would take additional system resources.

Also, virtually every stereo receiver in the world that can handle Dolby Digital can also handle Dolby Pro-Logic, but the reverse is not true.

[5] 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.

3.4 Audio Rules & Guidelines

3.4.1 Streaming Audio Buffer Size

Streaming audio plays back at a rate of 128,000 bytes per second. It uses a split buffer system where one half of the buffer contains the audio that is currently playing, while new audio data is being read into the other half. Every time that the playback pointer reaches the half-way point or the end point, the library issues a disc read request to get more data for the half of the buffer that just finished playing.

The NUON can read streaming audio data from a DVD disc much more quickly than it plays it back. This means that theoretically, your streaming audio buffer can be fairly small, as long as it can keep ahead of the reading process. However, this doesn't usually work well in practice for two reasons. First of all, the library will attempt to retry a data read if an error occurs. So if you have a very small buffer, there will be no time to recover from such an error before you experience a playback glitch. Second, it's not very efficient to continually issue a stream of data reads for just a few sectors each. When you read blocks of 200-300 sectors or more, you can probably get throughput of about 1mb per second or better. But if you read blocks of just 5-10 sectors at a time, you might only get 30kb at a time.

You should usually make your streaming audio buffer as large as possible. A reasonable buffer size is between 256kb and 1mb. In terms of playback time, this represents between 2 and 8 seconds.

When you have a larger buffer, the library is able to read ahead of the playback pointer and this enables it to recover from minor disc errors that may occur. The idea is similar to the anti-shock mechanisms found in portable & automobile CD music players.

The other thing to consider is that if you have a reasonable disc buffer, it becomes possible for your application to access other data from the DVD while streaming audio is playing. As long as you are careful to read data in reasonable-size chunks, the read requests from the audio library and the read requests from your application will both have time to be processed.

3.4.2 SDRAM Buffer Management

The audio libraries use a portion of SDRAM as a work area. It is important to ensure that other parts of your program are not attempting to use this work area for other purposes.

A common mistake is to use fixed addresses in SDRAM space for things like frame buffers, texture maps, and so forth. This is only going to work so long as nothing else tries to do dynamic memory allocation. Since the audio library uses dynamic allocation when you call **AUDIOInit**(), it may receive a buffer that overlaps with the static assignments your program makes.

There are two ways around this problem. First, you can use **AUDIOInitX()** to initialize the audio library. This function requires you to allocate the SDRAM buffer yourself.

The second idea is to change your program to use dynamic memory allocation using the BIOS function **_MemAlloc()**.

4. Game Controllers

4.1 The Philosophy

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.

After reading that, you might think at first that we were trying to predict strange and bizarre controller types, the likes of which have never been seen before. While there may be room for a few of those in our specification, what we really had in mind was trying to anticipate minor variations of standard controllers.

For example, suppose you had a standard analog gamepad that also has a throttle wheel. How does the program know about the throttle wheel?

4.1.1 Why Arbitrary Controller ID Codes Are BAD

With most game consoles, the system assigns an arbitrary ID number to each controller type. When a game sees that port A has a controller of type $21 attached, it knows that controller type $21 has certain capabilities, and it can act accordingly. But what happens if the controller type code comes up as $33 instead? The program might not know that code and would therefore it would not be recognized, even though the controller may only be slightly different.

Without knowing the specific attributes of a controller, even if the program is capable of reading the controller data, it can't be sure how to interpret it. It has 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.

4.1.2 Codes For Specific Controller Attributes

Instead of assigning static ID numbers, the NUON BIOS allows an application to determine what capabilities are available on each controller. Each controller reports a special *properties* field to the BIOS that indicates which features are present. By looking at which features are indicated, or in some cases which combinations of features, the program can determine what controller features are available.

The NUON BIOS knows about game pads, analog flight joysticks, steering wheels, paddle controllers, computer mouse controllers, fishing reel controllers, and more. There are over twenty distinct attributes which can be combined in various combinations.

Not all controller combinations are legal, and the NUON BIOS doesn't necessarily recognize every possible legal variation. However, it does have built-in support for dozens of the most common and most likely controller variations. Furthermore, this can be extended by applications should that prove necessary in the future.

4.2 Multi-Controller Support

The NUON system supports up to eight wired controllers by use of a multi-controller adapter. The main console has two NUON ports labeled. Each port can accept a controller or other peripheral directly, or it can support a multi-controller adapter with 4 slots.

4.3 Infrared Remote Control

In addition to controllers connected via the NUON port, NUON applications are also capable of reading the system infrared remote control (IRC) device. This is always marked as being "available" and can be used in the event that no wired controller is connected.

The NUON BIOS will automatically map certain IRC buttons to the controller data structure's button fields as though it were a wired controller. The controller *properties* field will indicate which controller features are emulated.

Please note that an IRC is generally not a good choice for any game where reaction time is critically important. It is slower to respond and generally harder to use in most game play situations. Therefore, VM Labs recommends that applications assign all available wired controllers to individual game players before assigning the infrared remote control.

However, the above not withstanding, VM Labs strongly recommends that all applications attempt to be at least minimally playable using an IRC. With that in mind, please be aware of the guidelines below.

- In addition to those buttons which are otherwise mapped, you may also detect most of the regular IRC buttons through the controller data structure's *remote_buttons* field.
- In order to improve handling, it is often desirable to map multiple buttons to the same function. For example, while the "A" button may already be mapped to the "Fast Forward" button, it may be a good idea to also map the same function to the "3", "6" and "9" buttons of the numeric keypad.

Also, note that different IRC devices may behave differently when buttons are pressed under different circumstances, such as:

- When more than one button is pressed at a time. Some IRC devices may output a signal only for the most recently pressed button, while others may output a signal only for the button pressed first. On others, pressing a second button while the first is still held down will result in the IRC halting any output until all buttons are released and then a single button pressed again.
- When a button is held down. Some IRC devices may send a stream of repeating values, while others may send an initial value and then stop. Some may send an initial burst, then pause, then send a stream until the button is released.
- When a button is pressed repeatedly. Most IRC devices require that a certain length of time go by before a new button press is recognized. If you press a button again too soon, it's ignored. This delay period is going to be different from one device to another.
- When you tap and release a button. Some IRC devices will register the button as "pressed" for a certain minimum length of time regardless of how fast you release it. Others may require that the button be held down for a certain minimum time period before it registers.

Because of these differences between IRC devices, you should probably avoid customizing your application's controller handling for a particular IRC. If you do that, you're probably just making it worse for another IRC.

On the other hand, if your application can allow the controller to be adjusted in a more general fashion, this may allow the user to customize the response for whatever IRC may be used.

5. Media Access & Program Loading

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 that have been seen on other console systems.

5.1 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.

5.2 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.

5.2.1 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 disk read cache available.

5.2.2 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***.

5.3 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. It is included in the NUON SDK as **datatool.exe** in the **BIN** directory.

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.

5.3.1 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.

5.3.1.1 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.

5.3.1.2 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.

5.3.1.3 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.

5.3.1.4 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.

5.3.1.5 Creating Your Data File

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

5.3.1.6 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
_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.

5.3.1.7 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 5.3.1.6 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.

5.3.1.8 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

5.3.1.9 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.

5.4 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 or Macintosh, or other systems. So long as the NUON directory is present, the disc may contain any other data as needed.

5.4.1 Program Authentication

A program that is intended to load from DVD and 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.

This applies to the **NUON.RUN** program that is automatically executed, and also to any individual program COFF file that is intended to be loaded via the **LoadGame()** BIOS function.

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. Program authentication services are available only to commercial developers.

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.

5.4.1.1 Running Non-Authenticated Programs

Authentication is not required for programs running on the NUON development system hardware. You can create your own bootable discs on DVD-R and run them on the NUON development system without going through the authentication process. However, such discs will not function on a production consumer NUON DVD player.

Once a program is running, it is possible for it to load a non-authenticated program file into memory using the standard media access functions. Once the image of a COFF file is resident in memory, the BIOS function **_MemLoadCoff**() can be used to parse it and load the individual program sections to the appropriate memory addresses. However, because the entire program file must be resident in memory at the beginning of the process, there are more restrictions regarding program file size and memory usage than for the **LoadGame()** function. However, this method is ideally suited for many purposes, including code overlays.

5.4.2 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.

5.4.2.1 NUON Folder

This folder should contain the following files:

- 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.

Other data files or program files may be included as needed.

5.4.2.2 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 DVD authoring tools.

5.5 Making a NUON CDROM

Newer models of NUON DVD players support the ability to boot a program from CDROM (including CD-R or CD-RW). [6] This is the mechanism that allows users of the NUON Open Platform SDK to create their own bootable program discs.

Creating a bootable NUON CD is easy. Take a standard NUON program COFF file[7] and use the **createnuoncd** tool to create a file named “**NUON.CD”**. Then simply create a CD with this file in the root directory of the disc. The disc must use the ISO9660 format.[8] The Joliet extensions may be used but are not required. The disc may not be multi-session.

The NUON.CD file does not have to be specially authenticated in the same way that a program that boots from DVD. The **createnuoncd** tool will do everything required to convert your original program file into a valid bootable file. The resulting disc should play in any standard NUON DVD player that supports the use of CD-R discs.

Other files may also be located on the CD, but they will not be accessible to the program. The system code that is used to load the **NUON.CD** program from CDROM is specialized for that purpose. General access to CDROM files is not supported at this time. However, it is expected that within the next several months, VM Labs will be able to provide either a BIOS update or linkable library that will provide such support.

Once a program has been loaded from CDROM, it is possible for that program to eject the disc so that a DVD or DVD-R disc may be placed in the drive. Then the program can open files in the NUON directory as if it had booted from the DVD disc instead of CDROM. See the NUON BIOS documentation for information about how to control the disc tray.

[6] This cannot be done with older models because CD-R and CD-RW discs are not compatible with the DVD drive mechanism. In particular, the Samsung N2000 and Toshiba SD-2300 do not support CD-R and CD-RW discs.

[7] Please note that there are some restrictions regarding maximum program size and memory usage. Please see the online documentation included with the NUON SDK for more information.

[8] Note that CD-RW discs typically use the UDF file system when your system is configured so that you simply drag and drop files onto the disc. To use the ISO9660 file system, you may have to burn the disc as though it were CD-R instead of CD-RW.

6. 2D Graphics

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

6.1 M3DL Library

The M3DL library has a variety of basic sprite functions.

6.2 MML2D Library

MML2D is designed to provide basic 2D graphics functions for rectangles, ellipses, and lines in a variety of styles. It also provides support for text output using TrueType outline fonts.

6.3 AlphaMask Graphics Library

The AGL is a C++ class library which provides a wide range of 2D graphics rendering functions for geometric shapes, lines, gradient filled areas, text, and much more. See the AGL documentation for more information.

6.4 DMA Blitting

An application can also use the DMA controller itself to render graphics. The SDK includes examples for simple bitmap image display and simple sprites that use simple DMA code.

7. 3D Graphics

There are numerous ways to create 3D graphics on NUON. So let's start out by categorizing the various options.

7.1 M3DL Library

The M3DL library provides 3D transform functions and polygon rendering. Lighting support is not built in, but can be implemented at the application level using the provided matrix math functions.

7.2 mGL Library

The mGL library is patterned after a subset of the popular OpenGL library.

7.3 MML3D Library

The MML3D library is not as full-featured as M3DL or mGL, but it provides the basic shell of a 3D library and is a good place to begin writing your own 3D renderer.