V M L A B S

NUON™ BIOS Documentation

Revision 2.00.87

November 5, 2001

VM Labs, Inc.
520 San Antonio Rd
Mountain View, CA 94040
Tel: (650) 917 8050
Fax: (650) 917 8052

Copyright notice

Contents

1. Introduction

The NUON BIOS is a set of functions, interrupt service routines, and device drivers that are designed to abstract the hardware of a NUON based system and make it easy to port applications between different NUON systems (such as set top boxes and DVD based systems), including systems that will be based on more powerful processors.

1.1 NUON Boot Sequence

Below we describe the events that take place when a NUON system first powers on. The exact details of which component (BIOS or DVD player software) determines the disc type may vary from system to system, but the overall boot flow will be similar.

1.1.1 All Systems

Hardware initialization, including loading device drivers from devices on the serial device bus.

1.1.2 DVD Based Systems

Check the type of disk (DVD or CD-ROM). If it is a DVD, use the Micro-UDF file system (as specified in OSTA UDF spec, revision 1.50; see the OSTA web page[1].

CD-ROMs are not, in general, bootable, although the CD/VCD player may be able to read files from the ISO9660 file system on a CD-ROM or multisession CD and use those files for special purposes (like extending the built-in VLM player).

Once the file system type has been determined, look for a `NUON` directory at the root.

1. If `NUON` is not found, load and run DVD, Video CD, or CD-DA decode software from ROM, as appropriate.
2. Otherwise: load and display `NUON.N16`. This is a 352x288 pixel, 16bpp image stored in raster order. The display buffer is at the base of main-bus RAM. If the system is displaying NTSC video, only the middle 240 lines will be displayed.
3. Validate the disk. If the disc is not properly authenticated, reboot.
4. Load the file `NUON/NUON.`*XXX* (if it exists) into system RAM, and cause MPE 3 to start executing it. This is a *model-specific* executable, useful for firmware-modifying executables, and in cases where the built-in firmware in a model is unable to run a particular application without large-scale changes. See section 18 for the *XXX* codes associated with particular models.
5. Load the file `NUON/NUON.RUN` (if it exists) into system RAM, and cause MPE 3 to start executing it.

[1] http://www.osta.org

When the game finishes, it calls the **_BiosExit** function, which resets the system software to look for a new disk.

Note the following requirements for a DVD based system:

1. All files must be contiguous on the disk.
2. It is possible to make a disk that plays a movie on a regular DVD player but plays a game on NUON.
3. Movies in the `VIDEO_TS` subdirectory can be played on a regular DVD; movies in the `NUON` directory will only play on NUON enhanced players.

1.1.3 Set Top Systems

Start running the MPE 3 server (the "dispatcher") that listens to requests from the PowerPC (or other host processor) for native OS (like Maui) or MML services.

To start a NUON-only game, the PowerPC puts the code (plus headers) into system RAM and then sends a "Start Program" request to MPE3. MPE3 then arranges to validate the code and (if it passes) to start running at that address. The game then takes over control of the system. When it is finished, it calls the **_BiosExit** function, which notifies the host that NUON is finished and re-starts the MPE 3 server.

On the host side, an application will have to be available that listens for network requests and/or requests for keyboard and similar I/O from NUON, and that sends back the appropriate data to the requesting MPE. This application might be running at all times (if the STB has a multitasking OS and if this makes sense) or it may run only for as long as the NUON game is running, exiting when it receives notification from the NUON BIOS that the game is finished.

1.1.4 Hybrid Systems

Follows the boot sequence for set top systems. The host processor (e.g. PowerPC) may send a command to the MPE 3 dispatcher asking it to check the drive. In this case, the DVD drive is checked, and if a valid disk is there then a response (either "normal DVD" or "NUON bootable DVD") is sent back to the host. The host may then either send a "start DVD movie playback" or "boot NUON disk" command. In the latter case, the application is loaded, verified, and run from DVD as in a DVD based system (section 1.1.2).

1.1.5 What environment does the booted program see?

1. The program starts in MPE 3.
2. The program starts up with 4K iram cache/3K dtram cache, with 512 bytes of dtram reserved for local ram functions. The remaining 512 bytes of dtram is available for user code and stack. The cache is valid, usable, and sync'd with memory.

3. The BIOS is set up for cached operation, i.e. the level 1 isr is running out of cached memory. The video isr, the timer 0 isr, and the comm receive isr are running; no other isrs are active at boot up time.

4. MPE 0 is running local code to support disk operations and streaming PCM audio. MPE 3 is responsible for loading up any other MPEs it wants to run.

5. The first 64K of system RAM and the last 640K of system RAM is reserved for BIOS use. If the Presentation Engine is in use, the last 3200K of system RAM is reserved for the BIOS and Presentation Engine. All other system RAM and main bus RAM is usable by the program. At initial boot time, the video is set up to display a frame buffer at the base of main bus RAM, containing the contents of the `NUON/NUON.N16` file.

1.2 BIOS Compatibility Guidelines

Applications must obey the following rules in order to ensure that they will operate properly with the BIOS.

1. Applications may use up to 8 MB of main bus RAM and up to 8MB - 704K of system RAM.

2. The first 64K of NUON-visible system RAM is reserved for the BIOS.

3. The last 640K (beginning at address `0x80760000`) of NUON-visible system RAM is reserved for the BIOS.

4. For software that uses Presentation Engine functions, the last 3200K of NUON-visible system RAM (beginning at address `0x804e0000`) is reserved for the BIOS and Presentation Engine.

5. The following hardware may be accessed only through the BIOS:

   (a) video
   (b) MPE 3 cache configuration
   (c) controllers (and other serial bus devices)
   (d) media (DVD/memory cards/etc.)
   (e) network and modem
   (f) hardware timers

   The audio hardware may be accessed only through the MML audio libraries, or through VM Labs approved substitutes.

   Note that there are some standard C library functions that access the above hardware (for example, the C library **open** and **read** functions). These will end up calling the BIOS, indirectly.

6. Only MPE 3 may access the BIOS. It is also subject to the following restrictions:

   (a) It must not access the communication bus directly. All comm bus accesses must be via the BIOS, so that comm bus based devices (such as controllers) will work correctly.
   (b) It must not use interrupt driven DMAs. (This is generally true of cached MPEs, because cache misses cause DMAs).
   (c) Interrupt vectors may be set only via the **_IntSetVector** BIOS call.

7. MPE 0 is also subject to some restrictions, in environments where media layer functions or streaming audio or video are desired.
   (a) It must not use the caches.
   (b) Internal memory available to the program is only 4k each of instruction and data memory.
   (c) It must not access the communications bus directly. All comm bus accesses must be via the mini-BIOS.

8. Playing back MPEG2 movies is a special case. The normal BIOS routines for MPEG playback will basically take over the system while the movie is being played (this will be suitable for simple movie clips during games). For more sophisticated uses of MPEG2 (e.g. electronic program guides) VM Labs will provide a library for developers to link against, so that their application can continue to run while movies are playing. Obviously there will be severe restrictions during this mode. The exact details of this library are still being worked out.

2. BIOS Configuration and Control

2.1 _BiosGetInfo

`#include <nuon/bios.h>`
struct BiosInfo * **_BiosGetInfo**(*void*)

This function returns a pointer to a structure which gives some information about the BIOS. This structure is guaranteed to have the following fields:

`major_version` BIOS major version number. This changes only for major new BIOS releases.

`minor_version` BIOS minor version number. This changes for major bug fixes or whenever a new function is added.

`vm_revision` VM Labs revision number. This changes for every release (unless one of the other fields changes!)

`oem_revision` Reserved for OEM use.

`info_string` A zero terminated ASCII string giving some information about the BIOS. This will include a formatted copy of the version number information above, as well as a vendor name and copyright notice.

`date_string` A zero terminated ASCII string giving the date of the BIOS build. This field is much less useful than the various version fields above, since the same BIOS version could be built on different dates.

2.2 _GetSystemSetting

`#include <nuon/sysinfo.h>`
int **_GetSystemSetting**(int *select*, void * *data*, int *len*, int *flags*)

Gets information about a system setting. *select* selects which setting to retrieve. *data* is a memory location into which the setting will be written (if it will fit). *len* is the amount of space available at *data*. *flags* determines whether the current setting (if *flags* is 2) or system setting (if *flags* is 1) will be read. These will normally be the same, although the current setting may be changed by the application.

Most system settings are 1 byte in length, but the exact length required may be obtained from the **_GetSystemSettingLength** call.

_GetSystemSetting returns 0 if it is successful, otherwise it returns an error code. Possible errors include `EINVAL` if *select* is not valid, or `ENAMETOOLONG` if the requested setting will not fit in *len* bytes.

Possible values for *select* include:

Value	Meaning
`kPlayerLanguage`	Default language used by the DVD player (presently may be one of `nvlNone`, `nvlPlayerEnglishLanguage`, `nvlPlayerFrenchLanguage`, or `nvlPlayerSpanishLanguage`).
`kAudioOutChannels`	Bitmask of audio output channels.
`kParentalCountryCode`	2 byte country code (in ASCII).
`kParentalLevel`	Current level of parental lockout (default is `0xf`, unlocked).
`kRegionCode`	Region code of this DVD player, or 0 if this is not a DVD player and has no similar notion of region codes.
`kDisplayAspectCode`	Aspect ratio of the television; 0 for 4:3, 1 for 16:9.
`kTvSystem`	Current TV output setting: 0 for NTSC, 1 for PAL, 2 for PAL60
`kSupportedVideoMaterial`	Bitmask specifying the type of video material supported Bit 0 - NTSC Bit 1 - PAL
`kSupportedVideoSystem`	Bitmask specifying the type of video output standard supported Bit 0 - NTSC Bit 1 - PAL Bit 2 - PAL60

Besides the values above, *select* may also be set to `kGameCookie` plus a game specific id (which must be assigned by VMLabs). In this case, a 2 byte game specific value stored in the BIOS ROM may be retrieved. This provides a mechanism to allow games to configure themselves on future hardware. Maintainance and interpretation of the game cookies is entirely up to game developers; the BIOS merely reports the value which the developer assigned to the cookie for this class of machine.

2.3 _GetSystemSettingLength

`#include <nuon/sysinfo.h>`
int **_GetSystemSettingLength**(int *select*, int * *len*)

Sets the integer pointed to by *len* to the length of the system setting required by *select*. See **_GetSystemSetting** for further information. Returns 0 if successful, or `EINVAL` if *select* is not a valid system setting on this machine.

2.4 _SetSystemSetting

`#include <nuon/sysinfo.h>`
int **_SetSystemSetting**(int *select*, void * *data*, int *len*, int *flags*)

Changes the current setting of a system setting. The parameters *select*, *data*, and *len* have similar meanings to their use in **_GetSystemSetting**, except that *data* points to the data to be written. *flags* must be 2; only the current settings may be changed by games.

Returns 0 if successful, `EINVAL` if *select* is not valid, or *ENAMETOOLONG* if *len* doesn't match the value returned by **_GetSystemSettingLength** for *select*.

2.5 _LoadSystemSettings

`#include <nuon/sysinfo.h>`

int **_LoadSystemSettings**(*void*)

Loads all system settings from NVRAM. This may be called by applications to refresh settings, although typically this will not be necessary (the default settings provided by the boot system should be used).

2.6 _LoadDefaultSystemSettings

`#include <nuon/sysinfo.h>`

int **_LoadDefaultSystemSettings**(*void*)

Reset all the system settings to default values, regardless of what values are saved in NVRAM. This will very rarely be needed; the **_LoadSystemSettings** call, which uses the values saved in NVRAM, should normally be used instead.

2.7 _BiosInit

`#include <nuon/bios.h>`

void **_BiosInit**(*void*)

This function initializes the BIOS. The C run-time library causes **_BiosInit** to be called automatically whenever an application starts, so normally it will not be necessary for applications to make this call.

Note that only one MPE may call **_BiosInit**. After the call to **_BiosInit**, no other MPE may make this call until **_BiosExit** has been called. Also note that the current implementation of the BIOS requires that only MPE 3 may make this call.

2.8 _BiosExit

`#include <nuon/bios.h>`

void **_BiosExit**(int *exitcode*)

Applications must call this function when exiting. This will cause the appropriate main application (the DVD gui interface for DVD applications, or host processor server for set-top boxes) to be restarted.

The C library **exit** and **_exit** functions will call this function automatically, as will the standard C runtime initialization code, if `main()` returns.

It is the application's responsibility to cleanly shut down all MPEs in use and to issue **_MediaClose** calls for all handles returned from **_MediaOpen** before calling **_BiosExit**.

On development systems, **_BiosExit** will cause the processor to halt. On production systems, it will either cause the system to reboot or else it will cause a player shell (such as the one used on hybrid discs) to be reloaded and restarted, as appropriate.

2.9 _BiosReboot

`#include <nuon/bios.h>`
void **_BiosReboot**(*void*)

Reboots the system. This should be used only at the user's explicit request, and even then only in exceptional circumstances. Needless to say, this function does not return.

2.10 _BiosPoll

`#include <nuon/bios.h>`
int **_BiosPoll**(char * *msg*)

This function should be called by applications at periodic intervals (at least 5 times per second) to check for special events, such as emergency broadcast messages. The function also causes watchdog timers to be reset on systems that have them, which means that failure to call **_BiosPoll** could result in the system rebooting after 200 milliseconds have elapsed since the last **_BiosPoll** call.

The parameter *msg* should point to a space containing at least 256 bytes of memory. This memory may be overwritten with a (zero terminated) ASCII string describing a message from the BIOS to be displayed to the user.

_BiosPoll returns 0 if there are no pending events. Otherwise it may return one of the following non-zero values:

Value	Meaning
`kPollDisplayMsg`	The application should display the ASCII text message placed in *msg* to the user. If for some reason the application is unable to do so, it should pause and call **_BiosPauseMsg** to display the message for it.
`kPollPauseMsg`	The BIOS wishes to display a message for the user. The application should call **_BiosPauseMsg**, passing the value returned from **_BiosPoll** (`kPollPauseMsg`, in this case) and *msg* as parameters.
`kPollSaveExit`	An event has occurred that requires the application to exit immediately. If possible, the application should save its current state before exiting.

If for any reason **_BiosPoll** returns a value other than one of those listed above, the application should pause and call **_BiosPauseMsg** with the parameter *rval* being the value returned by **_BiosPoll** (similar to what should be done if `kPollPauseMsg` is returned).

Use of **_BiosPoll** to monitor for messages is not enforced on most DVD systems, but it may be required for applications to function properly on set top boxes.

2.11 _BiosPauseMsg

`#include <nuon/bios.h>`

int **_BiosPauseMsg**(int *rval*, char * *msg*, void * *framebuf*)

Applications must call this function when **_BiosPoll** has indicated that there is a message pending that must be displayed to the user. The call to **_BiosPauseMsg** must occur within 50 milliseconds after the return from **_BiosPoll**, or results are unpredictable. *rval* is the value previously returned from **_BiosPoll**; typically this will be `kPollPauseMsg`, but could be any other non-zero return value from **_BiosPoll**. *msg* points to the the 256 byte buffer passed to the **_BiosPoll** function; **_BiosPoll** had filled it in with information that **_BiosPauseMsg** will use.

framebuf is a pointer to an area of at least 345600 bytes of SDRAM, aligned on a 512 byte boundary, that the BIOS may use as a frame buffer for the message to be displayed. The call to **_BiosPauseMsg** may take an arbitrary amount of time to complete, so the application should enter a pause mode before making this call. After the **_BiosPauseMsg** call returns, video is in an undefined state – the application should use the appropriate BIOS video calls to reset the screen to its own frame buffer.

_BiosPauseMsg may return one of the following values:

Value	Meaning
`kPollContinue`	The application should continue normally.
`kPollSaveExit`	An event has occurred that requires the application to exit immediately. If possible, the application should save its current state before exiting.

2.12 _PatchJumptable

`#include <nuon/bios.h>`

void * **_PatchJumptable**(void * *entry*, void * *function*)

Patches the BIOS jump table. *entry* is the label corresponding to the BIOS function to be patched. *function* is the new function to be called to perform the requested BIOS service. The return value is the address of the old function performing the BIOS service (so chaining is possible).

For example, to patch the **_BiosExit** function to call the function `NewExit`, do:

```
OldExit = _PatchJumptable(_BiosExit, NewExit);
```

2.13 _CompatibiltyMode

`#include <nuon/bios.h>`

int **_CompatibilityMode**(int *mode*)

Sets the processor caches and speed into one of various modes. If *mode* is 0, the caches and speed are set up for compatibility with Aries 2 applications. If *mode* is 1, the caches and speed are set up for Aries 3 operation, if possible (if the processor is actually an Aries 2, then it will of course remain in Aries 2 mode). If *mode* is anything else, results are unpredictable.

Applications will normally be started by the system in Aries 2 mode, and should switch to higher speed modes if they are capable of it.

The mode value in which the system was operating at the time of the call is returned. If this is `-1` then the system is unable to change modes (and almost certainly is an Aries 2 system).

3. Interrupt Service Routines

The level 2 interrupt service vector is used by the BIOS, and so is NOT available for use by applications.

Level 1 interrupt service routines (ISRs) may be set up by the programmer for a variety of conditions. The BIOS provides a level 1 interrupt service routine that saves registers and checks the `intsrc` and `inten1` registers to see which interrupts should be serviced. It then vectors through a table for these interrupts. Programmers may modify the interrupt vector table with the **_IntSetVector** function.

Programmer installed ISRs should follow the following conventions:

1. They may use registers `v0`, `v1`, `v2`, `r29`, `rc0`, and `rc1` as scratch registers, and need not preserve condition codes (the condition code register and `rzi1` are saved by the BIOS level 1 interrupt routine). All other registers must be preserved.

2. Any programmer installed ISR should return via a normal `rts` call (using register `rz`) to the BIOS's main interrupt processing loop.

3. The code for programmer installed ISRs may reside either in a cached memory space or in the local instruction RAM (section `intcode`).

4. The BIOS is not re-entrant. Only a limited number of BIOS functions may safely be made from an ISR; see the section on re-entrant calls (section 3.3).

5. It is the responsibility of the ISR to clear the interrupt condition. This must be done inside the ISR because different interrupts may require that the clearing be done at different times (pulse interrupts require different treatment than level interrupts). Therefore, user ISR functions should contain a `st_io` instruction that sets the appropriate bit in the `intclr` register. There are three exceptions to this: the interrupts reserved for the BIOS and marked with a (*) in the table below are handled by both the BIOS and the user ISR code, and in this case the BIOS clears the interrupt at the appropriate time. These interrupts are the `kIntrCommRecv`, `kIntrVideo`, and `kIntrSystimer0` interrupts. See the discussion below for more information on these.

3.1 _IntSetVector

`#include <nuon/bios.h>`
void * **_IntSetVector**(int *which*, void * *newvec*)

Installs a level 1 interrupt service routine. The parameter *which* specifies which interrupt vector is to be modified. This is the same as the bit number of the interrupt in the `intsrc` and `inten1` registers (see the table below).

newvec is a pointer to the function to be executed when the interrupt occurs. This may be either in internal (uncached) instruction memory, or in cached memory. If *newvec* is `NULL`, then this interrupt is disabled.

_IntSetVector usually returns the address of the current interrupt service routine for this interrupt, or `NULL` if no ISR has been installed before for this interrupt. However, see some exceptions below.

Interrupt Number	Symbolic Name	Meaning
31	`kIntrVideo`	VDG beam position interrupt (*)
30	`kIntrSystimer1`	system timer 1
29	`kIntrSystimer0`	system timer 0 (*)
28	`kIntrGPIO`	GPIO IO combined interrupt
27	`kIntrAudio`	audio output interrupt
26	`kIntrHost`	external host processor interrupt
25	`kIntrDebug`	debug control unit interrupt
24	`kIntrMBDone`	MCU macro-block done interrupt
23	`kIntrDCTDone`	MCU DCT done interrupt
20	`kIntrIIC`	serial peripheral bus interrupt
16	`kIntrSystimer2`	system timer 2
5	`kIntrCommXmit`	comm bus transmit buffer empty interrupt
4	`kIntrCommRecv`	comm bus receive buffer full interrupt (*)
1	`kIntrSoftware`	software generated interrupt
0	`kIntrException`	local MPE exception

Note that since the BIOS must run on MPE 3, interrupts that are specific to other processors cannot usefully be captured with this mechanism, and so are not listed in the table above. Similarly, the DMA interrupts may not be used on a cached processor.

All interrupt handlers are called via an ordinary `jsr` instruction, and so should return with `rts`. The top level interrupt handler saves vector registers `v0`, `v1`, and `v2`, as well as the counter registers `rc0` and `rc1`, and general purpose register `r29`. The comm bus registers and DMA registers are also preserved by the top level handler. The user provided ISR must save and restore any other registers it uses; typically this will be done with `push` and `pop` instructions, or by using the `r31` stack. See the discussion above (section 3) for a fuller list of requirements for programmer installed ISRs.

Please note that it is the responsibility of the interrupt handler to clear the underlying condition that caused the interrupt (for example, to clear the appropriate bit in the `intsrc` register). An exception is that interrupt sources reserved for the BIOS (those marked with a (*) above), for which the BIOS clears the interrupt condition appropriately before calling any user installed routines.

Several interrupts are reserved for the BIOS, and are marked with a (*) above. These are the VDG beam position interrupt, the system timer 0 interrupt, and the comm bus receive buffer full interrupt. If **_IntSetVector** is used to set one of these interrupts, then the user's interrupt vector will be called only after the BIOS has processed the interrupt first.

3.1.1 ***kIntrVideo***

Any application-provided `kIntrVideo` interrupt service routine will be called once per field, at the beginning of the vertical blanking interval.

3.1.2 ***kIntrSystimer0***

Any application-provided `SYSTIMER0` interrupt service routine will be called by the BIOS just after the normal BIOS timer 0 functions are called. This will occur at the BIOS clock rate, which is 200 Hz.

3.1.3 ***kIntrCommRecv***

The BIOS maintains a list of comm bus handlers. Calling **_IntSetVector** to set the `kIntrCommRecv` vector adds another handler. Each such handler should have the C prototype:
int **CommHandler**(int *id*, int *comminfo*, int *reserved1*, int *reserved2*, int *p0*, int *p1*, int *p2*, int *p3*)

id is the comm bus id of the sender of the packet; *comminfo* is the `comminfo` data associated with the packet (if any). *p0* through *p3* is the actual packet data. The BIOS calls each comm bus handler in turn. If a handler recognizes and deals with a packet, it should return -1, in which case the BIOS terminates processing of the packet (it does not call any more packet handlers). Otherwise, the handler should return with registers `r0` through `r7` unchanged. This latter requirement may not be satisfied by functions written in C, so it may be necessary to write the top level of the comm bus handler in assembly language.

Please note that all comm bus handlers must obey the C calling conventions, and due to the structure of interrupt code on some platforms they must keep `r31` available for use as a stack pointer at all times, and must not modify `acshift` at any point. C code will automatically follow these restrictions.

The final handler in the BIOS chain of packet handlers is the comm bus queue function that queues packets for later reception by **_CommRecvInfo**.

Note that when **_IntSetVector** is used to install a comm bus handler, it returns a magic value that may be passed again to **_IntSetVector** to uninstall the handler. This is somewhat different from the normal behavior of **_IntSetVector**, but is useful in this case because more than one packet handler may be active at a time.

3.2 _IntGetVector

`#include <nuon/bios.h>`
void * **_IntGetVector**(int *which*)

Retrieves the current interrupt service routine for the interrupt number specified by *which*. See the **_IntSetVector** function for a description of what interrupt service routines there are, and also how to change them.

If no interrupt service routine has been established for *which*, then **_IntGetVector** will return `NULL`.

3.3 Re-entrant System Calls

Most BIOS functions are not re-entrant, and so they may not be called from inside an interrupt or from code running on another MPE (for example, code started via **_MPERunThread**). Re-entrant functions are listed below.

Not all re-entrant functions are safe to call from another MPE. Many BIOS services rely on interrupts that are only enabled in MPE 3, or on data structures which may be in MPE 3's cache (and not yet written out to memory).

3.3.1 Calls which are both thread-safe and callable from another MPE

The only BIOS functions which may be called from another MPE are **_CommSend** (and the related **_CommSendDirect** and **_CommSendInfo**), **_DMALinear**, **_DMABiLinear**, **_DCacheSync**, **_DCacheSyncRegion**, and **_DCacheFlush**. Note that although **_CommSend** is safe to use from another MPE, none of the comm bus receive functions will work from another MPE (they rely on interrupts which happen in MPE 3).

3.3.2 Calls which are re-entrant but only callable from MPE 3

The **_IntSetVector** The **_MediaRead** function may be called from within a comm bus or timer interrupt service routine. Calling it from within other interrupt service routines may cause video glitches or other timing problems.

4. DMA Functions

4.1 _DMALinear

```
#include <nuon/dma.h>
```
void **_DMALinear**(long *flags*, void * *baseaddr*, void * *intaddr*)

Performs a main bus or other bus linear DMA transaction. Which bus to use is determined by the external address *baseaddr*.

flags are the DMA flags, as described in the “Main Bus” or “Other Bus” section of the NUON Specifications. These flags must be appropriate for a linear DMA. Typically *flags* will contain the number of scalars to transfer in bits 16 to 22, and bit 13 will be clear for a DMA write and set for DMA read. If any of the special main bus linear transfer command mode bits are set, then *baseaddr* must be in SDRAM (so that the main bus will be used).

baseaddr is the base address, which may be in SDRAM, system RAM, or ROM. *intaddr* is the internal address for the DMA, which must be in the MPE’s local RAM.

Note that the length of the transfer should not exceed 32 scalars (if the external address is known to be in SDRAM, 64 scalars is the limit). Longer transfers may be accepted by the hardware, but will interfere with interrupt latency and may cause I/O transfers to malfunction.

_DMALinear will not return until the DMA is finished.

4.2 _DMABiLinear

```
#include <nuon/dma.h>
```
void **_DMABiLinear**(long *flags*, void * *baseaddr*, unsigned long *xinfo*, unsigned long *yinfo*, void * *intaddr*)

Performs a main bus pixel mode or MPEG mode DMA transaction. *xinfo* gives the X length (in the high 16 bits) and position (in the low 16 bits), and similarly *yinfo* gives the Y length and position. The other parameters are the flags, base address, and internal address as described in the “Main Bus” section of the NUON Specifications. Note that the external memory address *baseaddr* must be in SDRAM.

Note that the total length of the transfer must not exceed 256 bytes. Longer transfers may be accepted by the hardware, but will interfere with interrupt latency and may cause I/O transfers to malfunction.

_DMABiLinear will not return until the DMA is finished.

4.3 Specialized DMA functions

These functions are not as general as **_DMALinear** and **_DMABiLinear**, and may not be as efficient on future machines. They should be used with care, as they are not as robust against hardware bugs as the other DMA functions.

4.3.1 _Dma_wait

void **_Dma_wait**(volatile int * *ctrl*)

Reads the DMA control register pointed to by *ctrl* (which should point to either `odmactl` or `mdmactl`) and waits until all DMA on that bus is complete.

4.3.2 _Dma_do

void **_Dma_do**(volatile int * *ctrl*, void * *cmdBlock*, int *waitFlag*)

Execute a DMA command block. *ctrl* must point to either `odmactl` for an other bus DMA, or `mdmactl` for a main bus DMA, as appropriate to the DMA command to be issued. *cmdBlock* must point to the DMA command block, which must be in internal MPE memory (such as memory returned by **_MemLocalScratch**). *waitFlag* should be 1 if the command is to block until the DMA is completed.

Note that hardware bugs mean that for all practical purposes *waitFlag must* be 1 for all main bus DMAs. Also note that if *waitFlag* is 0 then **_Dma_do** will return with no guarantee tha the command has even been read by the hardware yet; in this case, it is not safe to write into the memory pointed to by *cmdBlock* until after **_Dma_wait** has been called.

5. Comm Bus Functions

5.1 _CommSend

`#include <nuon/comm.h>`
void **_CommSend**(int *target*, long * *packet*)

Sends a communication bus packet to the destination whose communication bus id is *target*. *packet* points to the four long words to be sent. The `comminfo` value sent along with the packet will always be set to 0.

The **_CommSendDirect** function provides another interface which does not require the packet to be in memory, and hence may be faster for some purposes.

5.2 _CommSendInfo

`#include <nuon/comm.h>`
void **_CommSendInfo**(int *target*, int *info*, long * *packet*)

Sends a communication bus packet to the destination whose communication bus id is *target*. *packet* points to the four long words to be sent. *info* is an 8 bit quantity to be placed in the `comminfo` register. If *target* is an MPE, then this data is transmitted along with the packet and may be retrieved from the `comminfo` register on the destination MPE. If *target* is a hardware unit, then *info* is ignored.

The **_CommSendDirect** function provides another interface which does not require the packet to be in memory, and hence may be faster for some purposes.

5.3 _CommSendDirect

`#include <nuon/comm.h>`
void **_CommSendDirect**(long *p0*, long *p1*, long *p2*, long *p3*, int *target*, int *info*)

Sends a communication bus packet to the destination whose comm bus id is *target*. *p0* through *p3* are the 4 scalars of the packet to send. *info* is an 8 bit quantity to be placed in the `comminfo` register and transmitted along with the packet.

_CommSendDirect is generally the fastest way to send a comm bus packet, since it does not require the packet to be fetched from memory. However, if the packet is already in memory in an array of long words, use **_CommSendInfo** to send the packet instead.

5.4 _CommRecvInfo

`#include <nuon/comm.h>`
int **_CommRecvInfo**(int * *info*, long * *packet*)

Receives a single communication bus packet; the four long words of the packet will be placed in the memory pointed to by *packet*, and the 8 bit contents of the `comminfo` register will be placed in the memory pointed to by *info*. If the sender specified extra information (*e.g.*, if the packet was sent with the **_CommSendInfo** function) then this will be the extra info; otherwise, it may contain garbage.

_CommRecvInfo returns the comm bus id of the processor that sent the packet.

If no packet is available when **_CommRecvInfo** is first called, then it will wait until a packet is received. For a non-blocking read function (one that returns immediately if no data is available) use **_CommRecvInfoQuery**.

5.5 _CommRecvInfoQuery

`#include <nuon/comm.h>`
int **_CommRecvInfoQuery**(int * *info*, long * *packet*)

Attempts to receive a single communication bus packet. If a packet is available, then four long words of the packet will be placed in the memory pointed to by *packet*, and the 8 bit contents of the `comminfo` register will be placed in the memory pointed to by *info*. If the sender specified extra information (for example, if the packet was sent with the **_CommSendInfo** function) then this will be the extra info; otherwise, it may contain garbage. In this case **_CommRecvInfoQuery** returns the comm bus id of the processor that sent the packet.

If no packet is available when **_CommRecvInfoQuery** is called, then it will return immediately a value of `-1`.

For a blocking read function (one that waits for data to become available) use **_CommRecvInfo**.

5.6 _CommSendRecv

`#include <nuon/comm.h>`
int **_CommSendRecv**(int *target*, long * *packet*)

Sends a communication bus packet to the destination whose communication bus id is *target*, and then waits for a response. *packet* points to the four long words to be sent; on return these four long words are overwritten with the response received. The return value is the comm bus id of the sender of the response that was received, which will be *target*.

_CommSendRecv may lock out interrupts while it is running, so it should be used only to request data from hardware with low latency. Typically it would be used to read registers from hardware units such as the miscellaneous I/O controller that are accessible only via the comm bus.

5.7 _CommSendRecvInfo

`#include <nuon/comm.h>`
int **_CommSendRecvInfo**(int *target*, long * *packet*, int *info*)

Sends a communication bus packet to the destination whose communication bus id is *target*, *info* will be placed in the `comminfo` register, of which only the lower 8 bits is valid, and then waits for a response. *packet* points to the four long words to be sent; on return these four long words are overwritten with the response received. The return value is the comm bus id of the sender of the response that was received, which will be *target*.

_CommSendRecvInfo may lock out interrupts while it is running, so it should be used only to request data from hardware with low latency. Typically it would be used to read registers from hardware units such as the miscellaneous I/O controller that are accessible only via the comm bus.

5.8 Other comm bus functions

commpacket **_comm_recv**(*void*)

This function is not callable from C, but may be useful for assembly language programmers. It is similar to **_CommRecvInfo**, except that the packet is returned in vector register `v0`, the comm bus ID of the sender is returned in register `r4`, and the sent comm info value is returned in `r5`.

int **_comm_query**(*void*)

Checks to see if any comm bus packets are waiting to be returned by **_CommRecvInfo**. Returns 0 if there is a packet waiting, -1 if not.

This function is obsolete. It is much better to use the **_CommRecvInfoQuery** function, or to install a comm bus interrupt handler with **_IntSetVector**.

6. MPE Control and Execution Functions

6.1 _LoadGame

`#include <nuon/mpe.h>`
void **_LoadGame**(const char * *name*)

Loads and runs COFF file from disk or network. *name* is the name of the COFF file to load. It should be an absolute path name (*i.e.* it should include the full path to the file, such as `"/udf/nuon/nuon.run"`).

This function acts like a “chain” call; the currently executing program is completely replaced by the new program in the COFF file. This COFF file must be properly signed and authenticated (except that some development systems will allow **_LoadGame** on unauthenticated COFF files such as are normally produced by the linker).

It is advisable to shut down all user MPE activity and wait for all I/O to finish before making this call.

Note that **_LoadGame** will never return, even in the event of an error. For security reasons, if an error occurs in **_LoadGame** then the system will reset. For this reason it is a good practice to check for the existence of the file *name* before passing it to **_LoadGame**.

6.2 _MemLoadCoff

`#include <nuon/mpe.h>`
int **_MemLoadCoff**(int *mpe*, void * *coffbase*, int *flags*, void * *extra*)

Loads a COFF file from memory. Any local memory sections are placed into the indicated MPE *mpe*. **_MemLoadCoff**, if successful, returns the entry point indicated in the COFF file. If there is an error while loading the COFF file, **_MemLoadCoff** returns 0. Note that the error checking is not very robust, and a bad COFF file might be loaded without an error being reported but still not work correctly.

_MemLoadCoff may have side effects on other MPEs. In particular, it attempts to shut down the media MPE and allocate it to accelerate decompression and loading of COFF files.

coffbase is a pointer to the COFF file in memory. This should simply be a verbatim binary copy of a COFF file produced by the linker or assembler, or a compressed COFF file produced by the `zcoff` tool. It should not have any authentication header.

flags contols how the loading is performed, and whether or not to start the MPE. It is formed by or’ing together the following flags:

Flag	Meaning
`LOADFLAGS_RUN`	start the target MPE after loading the COFF file, and stop the current MPE (note the last part!)
`LOADFLAGS_SAVE_MPE`	preserve local memory in any MPE used for COFF file decompression; needed if the application hasn't properly allocated MPEs
`LOADFLAGS_NOMEDIA`	do not touch the media processor; note that this flag does not work properly in BIOS versions prior to 1.03.10

For most purposes the *flags* parameter should be set to `0`. `LOADFLAGS_RUN` may be used to start the new MPE up, but it has the (unfortunate) side effect of stopping the current MPE. This is usually not desired; typically applications will wish instead to leave this flag clear, and to do **_MPERun** on the target MPE using the entry point of the COFF (which is returned by **_MemLoadCoff**).

The `LOADFLAGS_NOMEDIA` flag should be added if the application has changed the normal media handling. The `LOADFLAGS_SAVE_MPE` flag should be necessary only in exceptional circumstances where the BIOS MPE management has been bypassed.

extra is reserved for future applications, and should always be set to `(void *)0`.

Be careful to place the COFF file to be loaded and the program doing the loading in separate places in memory; if the new program overwrites the old, and the old is still trying to run, there will likely be a crash.

_MemLoadCoff returns the entry point of the loaded COFF file, or 0 if the load failed. However, note that if the `LOADFLAGS_RUN` flag was set, **_MemLoadCoff** will never return.

6.3 _StreamLoadCoff

int **_StreamLoadCoff**(char * *name*, long *base*)

This function is used by the firmware to load device drivers and applications over the serial port. It is not intended for use by end user applications.

name is the full path and name of the file to be loaded. *base* is currently unused, and should be set to 0. (It is intended eventually that if *name* is NULL, then *base* will be the address of a COFF file in memory; however, that functionality may not work properly right now. Use **_MemLoadCoff** to load COFF files from memory.)

_StreamLoadCoff returns 0 on success and a negative error code on failure.

6.4 _MPEAlloc

`#include <nuon/mpe.h>`
int **_MPEAlloc**(long *flags*)

Allocates an MPE, with properties given by *flags*. Returns the number of the MPE allocated. If no MPE matching the flags can be allocated, returns `-1`.

Any combination of the following flags (made by logically OR'ing them together) may be used. If *flags* is 0, then any completely free MPE may satisfy the request. The MPE running the mini-BIOS (if any) is not completely free, and so it will not be returned unless the `MPE_HAS_MINI_BIOS` flag is explicitly given.

Flag	Meaning
`MPE_HAS_ICACHE`	the MPE has an instruction cache
`MPE_HAS_DCACHE`	the MPE has a data cache
`MPE_HAS_CACHES`	the MPE has both instruction and data caches
`MPE_DTRAM_8K`	the MPE has 8K of local data memory
`MPE_IRAM_8K`	the MPE has 8K of local instruction memory
`MPE_HAS_MINI_BIOS`	the MPE has a "mini" BIOS available; this may mean that some MPE resources are used by the BIOS. See the mini-BIOS section for details.

6.5 _MPEAllocSpecific

`#include <nuon/mpe.h>`
int **_MPEAllocSpecific**(int *mpe*)

Allocates the given MPE. If *mpe* has already been allocated, returns `-1`, otherwise returns *mpe*.

6.6 _MPEFree

`#include <nuon/mpe.h>`
int **_MPEFree**(int *mpe*)

Frees the specific MPE *mpe*, which must have been previously allocated by either the **_MPEAlloc** or **_MPEAllocSpecific** function. If *mpe* is already free, returns `-1`, otherwise returns `0`.

6.7 _MPEsAvailable

`#include <nuon/mpe.h>`
int **_MPEsAvailable**(int *flags*)

If *flags* is `0`, then this call returns the total number of MPEs in the system. If *flags* is `1`, then this call returns the number of free MPEs currently available for allocation. All other values of *flags* are reserved for future use.

6.8 _MPEReadRegister

`#include <nuon/mpe.h>`
long **_MPEReadRegister**(int *mpe*, void * *addr*)

Reads a register from an MPE. The MPE must not be running (otherwise this call could cause the other MPE to crash). *mpe* is the number of the MPE to read from, and *addr* is the address of the register to be read within that MPE. This address must be in MPE relative format, and so will typically lie in the `0x2050xxxx` address range. The value of the register will be returned.

6.9 _MPEWriteRegister

`#include <nuon/mpe.h>`
void **_MPEWriteRegister**(int *mpe*, void * *addr*, long *value*)

Writes a value to a register in an MPE. The MPE must not be running (otherwise this call could cause the other MPE to crash). *mpe* is the number of the MPE, and *addr* is the address of the register to be written within that MPE. This address must be in MPE relative format, and so will typically lie in the `0x2050xxxx` address range. *value* is the value to be written into that register.

6.10 _MPEStop

`#include <nuon/mpe.h>`
void **_MPEStop**(int *mpe*)

Stops a running MPE, and has no effect on an MPE which is already stopped (except that some bits in the control register of that MPE may be cleared).

6.11 _MPEWait

`#include <nuon/mpe.h>`
long **_MPEWait**(int *mpe*)

Waits for a running MPE to stop on its own. *mpe* is the number of the MPE to wait for. If the target MPE halts because of an exception, **_MPEWait** returns `-1`; otherwise it returns the contents of the MPE's register `r0` at the time it halted.

6.12 _MPELoad

`#include <nuon/mpe.h>`
void **_MPELoad**(int *mpe*, void * *mpeaddr*, void * *linkaddr*, long *size*)

Loads code or data into a (stopped) MPE. *mpe* is the number of the MPE into which the code or data is to be loaded. *mpeaddr* is the destination address (which is MPE relative, *i.e.* given as though the MPE were MPE 0). *void* *linkaddr is the address of the code or data in system memory; in other words, it's where the code or data was originally loaded. *size* is the size of the code or data to be loaded.

6.13 _MPERun

`#include <nuon/mpe.h>`
void **_MPERun**(int *mpe*, void * *entrypoint*)

Starts an MPE. *mpe* is the number of the MPE, and *entrypoint* is the initial program counter for the MPE; normally this should lie inside of instruction RAM. The application is responsible for loading the correct code and data into the target MPE before calling **_MPERun**. Typically this will be done with **_DMALinear**. The Miscellaneous Utility Library `libmutil` has functions which make the process of starting an MPE somewhat simpler, and it is recommended that applications use those functions rather than calling **_MPERun** directly.

If **_MPERun** is called on an MPE which is already running, the results are unpredictable and unlikely to be anything useful.

6.14 _MPERunThread

`#include <nuon/mpe.h>`
int **_MPERunThread**(int *mpe*, void * *func*, void * *arg*, long * *stacktop*)

Runs C code on a C capable MPE. *mpe* is the MPE on which to run the code, which must have been allocated with the appropriate flags for C capable MPEs (`MPE_HAS_CACHES`). *func* is the address of the C function to run, which should take one 32 bit argument. void * *arg* is the argument which will be passed to the function. *stacktop* is a pointer to the end of a (vector aligned) region of memory which will be used as the C stack for the new thread (the C stack grows downwards).

The thread will continue running on the new MPE until it executes a `halt` instruction, or until the called function *func* returns.

Note that C code running on another MPE may access only a very few BIOS services – most BIOS functions do not work on any MPE other than MPE 3. The only services which are guaranteed to be available are the ones discussed in the section on re-entrant calls (section 3.3). Also, a thread running on another MPE should beware of calling any C library functions, unless they are known to be re-entrant and thread-safe.

MPE threads may communicate with the main task via shared memory; however, because there is no automatic synchronization of the caches, the shared memory must either be accessed via DMA or else care must be taken to manually synchronize the caches on the two processors.

_MPERunThread returns `0` if it fails to start the thread (for example, if the requested MPE is unable to run C code), and `1` if it succeeds.

Note that the MiniBIOS (section 7) runs on MPE 0 by default, and this is the only MPE other than 3 which may run C code. So usually it will be necessary to make the **_MediaShutdownMPE** call to free MPE 0 before calling **_MPEAlloc** to allocate a C MPE (one with caches). It will then be necessary to call **_MediaInitMPE** again before any of the media functions can be used.

6.15 _MPEStatus

`#include <nuon/mpe.h>`
long **_MPEStatus**(int *mpe*)

Returns information about an MPE, such as whether or not it has caches, how big its internal memory is, and whether it is presently allocated. The flags in the returned value are the same as those passed to **_MPEAlloc**, except that there are two additional flags to indicate how (or whether) the MPE is in use:

Flag	Meaning
`MPE_ALLOC_USER`	the MPE was allocated by an application
`MPE_ALLOC_BIOS`	the MPE is being used by the BIOS

It is possible for an MPE to be in use by both the BIOS and an application; in this case the MPE is running the MiniBIOS (section 7) along with some user code.

Note that early versions of the BIOS did not implement the **_MPEStatus** call, and on such BIOSes the returned value will always be `-1`. The return value should be tested before any of the flags are assumed to be valid.

7. The MiniBIOS

Because cache misses can impose interrupt latencies longer than some DVD drives are able to tolerate, the BIOS must run drive control code in an uncached MPE. Thus, a portion of an uncached MPE is reserved for BIOS use. By default the BIOS installs a piece of code at the top of MPE 0 to control drives. This code is called the MiniBIOS. The NISE audio library also uses this space for streaming audio and PCM sound effects. The top 4K of both the instruction and data RAMs on MPE 0 are taken up by the combined audio and DVD drive reading code.

Applications may use the lower 4K of iram and dtram on MPE 0 for their own code, provided that code obeys the following rules:

1. The code must not use interrupts (both the level 1 and level 2 ISRs are reserved for the MiniBIOS and audio libraries). None of the interrupt registers may be touched by the code.

2. The code must not cause any cache misses (neither the instruction nor data cache may be used on the MPE running the MiniBIOS).

3. All accesses to the comm bus must be performed through the MiniBIOS jump table.

4. Because the comm bus must always be available for the DVD drive, the application must use software flow control for comm bus packets. If comm bus packets are sent to the MPE hosting the MiniBIOS when its internal comm bus queue is full, those packets will be lost.

5. The MiniBIOS uses the `comminfo` register to determine whether packets received are designated for the application for for the DVD control code or streaming audio code. Values of `0x80` through `0xff` are reserved for MiniBIOS use. Other MPEs which send comm bus packets to the MPE hosting the MiniBIOS must be sure to set `comminfo` to a value between `0x00` and `0x7f`.

6. If the code wishes to terminate, it may use the `halt` instruction to do so. The MiniBIOS catches the exception generated by this instruction and shuts the user code down gracefully.

If for any reason the application wishes to run code on the MiniBIOS MPE (by default MPE 0) which does not obey the above restrictions, it will have to shut down the MiniBIOS with the **_MediaShutdownMPE** call and refrain from making any drive accesses. To resume using the DVD drive, call **_MediaInitMPE**. Note that if MPE 0 is still allocated when **_MediaInitMPE** is called, the BIOS will attempt to allocate another MPE for drive control. Thus, to use MPE 0 for running C code while still accessing the drive, do:

```
_MediaShutdownMPE();               // free MPE 0
c_mpe = _MPEAlloc(MPE_HAS_CACHES); // allocate MPE 0
new_media_mpe = _MediaInitMPE();   // start new media MPE
```

```
if (new_media_mpe < 0) {
   /* no drive accesses possible! We can't run
      the thread after all... */
   if (c_mpe >= 0) {
       _MPEFree(c_mpe);
       _MediaInitMPE()
   }
} else {
   /* everything is OK; start the thread */
   _MPERunThread(c_mpe, myfunc, arg, top_of_stack);
}
```

7.1 MiniBIOS jump table

The MiniBIOS jump table is located just before the level 1 interrupt service routine on that MPE. The jump table may thus be accessed relative to the `intvec1` register, which allows us to change its position when the MiniBIOS is run on MPEs other than MPE 0.

Offset	Function
`intvec1 - 4`	Get number of jump table entries.
`intvec1 - 8`	`MINIcommrecv`
`intvec1 - 12`	`MINIcommrecvquery`
`intvec1 - 16`	`MINIcommsend`
`intvec1 - 16`	`MINIcommhook`

Thus, for example, to receive a comm bus packet the application running under the MiniBIOS would do something like:

```
ld_io intvec1,r0
nop
sub   #8,r0    ; point to MINIcommrecv
jsr   (r0),nop
```

The functions are explained below. All MiniBIOS functions may change registers `r0` through `r7` inclusive, and they preserve all other registers.

7.1.1 *MINIcommrecv*

Receives a single comm bus packet. The packet itself is returned in `v0`; the comm bus id of the sender is returned in `r4` and the `comminfo` value sent with the packet is returned in `r5`. `MINICommrecv` waits until a packet is received before returning.

Note that `MINIcommrecv` and `MINIcommrecvquery` will not function if the application uses `MINIcommhook` to override the default comm bus handler.

7.1.2 *MINIcommrecvquery*

Exactly like `MINIcommrecv`, except that if no comm bus packet is available the function returns immediately with `r4` set to `-1`.

Note that `MINIcommrecv` and `MINIcommrecvquery` will not function if the application uses `MINIcommhook` to override the default comm bus handler.

7.1.3 *MINIcommsend*

Sends the packet in `v0` to the comm bus id specified in `r4`. The `comminfo` value to send along with the packet is given in `r5`; if the target is another MPE, this value should be in the range `0x00` to `0x7f`.

7.1.4 *MINIcommhook*

Changes the default comm bus handler. `r0` should point to a new comm bus service routine, which will be called when comm bus packets arrive. The service routine will receive the packet id in `r0`, the `comminfo` value in `r1`, and the packet itself in `v1`. It may change `v0` through `v2`, but must preserve all other registers. The comm bus handler should not take more than a few milliseconds to run, or else streaming audio and other MiniBIOS functions may be disrupted.

`MINIcommhook` returns the old comm bus handler in `r0`.

Applications may use this call if they wish to handle comm bus packets in an interrupt driven fashion (rather than the polling provided by `MINIcommrecv`). Note that `MINIcommrecv` and `MINIcommrecvquery` will not function if the application uses `MINIcommhook` to override the default comm bus handler.

8. Cache Control Functions

In the NUON hardware, the data and instruction caches are implemented by causing references to external memory (*i.e.* the SDRAM or system memory, as opposed to on-chip memory) to be re-mapped to references to internal memory, and (if necessary) a DMA from the external memory to internal to take place. This caching mechanism is always active. However, logically there are times when one or both of the caches is not being used (when only internal addresses are used, so no external memory references take place). In this situation, the cache is said to be disabled. Note that it is not, in general, possible to mix cache accesses with accesses to the local memory that is used for the cache, since every time a cache miss occurs the local memory may be overwritten, and similarly changing local memory contents may cause unexpected changes to external memory contents. So either all accesses should be to local memory only (cache disabled) or there are cache accesses and no accesses to the portion of local memory being used for the cache (cache enabled).

The BIOS requires that a portion of internal memory must always be used for both instruction and data caches. In other words, the instruction and data caches must always be enabled in order to use BIOS services. The cache may be resized, but it must always be there.

8.1 _DCacheSync

```
#include <nuon/cache.h>
```
void **_DCacheSync**(*void*)

Synchronizes memory with the MPE's data cache (*i.e.* writes cache contents out to system memory). After this call is made all data is guaranteed to have been stored to memory. Since the data cache is not a write through cache, it is usually necessary to make either a **_DCacheSync** or **_DCacheFlush** call before asking another MPE to access memory that may have been modified via the cache. **_DCacheSync** differs from **_DCacheFlush** in that it does not invalidate the cache, and thus it is the preferred means of ensuring that data is in SDRAM if normal caching operation is to continue. However, note that if you wish to use the cache to access a variable that another MPE has set, you must use **_DCacheFlush** to mark the cache invalid.

See also the **_DCacheSyncRegion** call for synchronizing just a portion of the cache.

The obsolete name **_synccache** is a synonym for **_DCacheSync**.

8.2 _DCacheSyncRegion

```
#include <nuon/cache.h>
```
void **_DCacheSyncRegion**(void * *startaddr*, void * *endaddr*)

Synchronizes memory with the MPE's data cache. After this call is made all data that is stored between addresses *startaddr* and *endaddr* (with *startaddr* included in the region, but *endaddr* not), will be written out to memory. Since the data cache is not a write through cache, it is usually necessary to make either a **_DCacheSyncRegion** or **_DCacheSync** call before asking another MPE to access memory that may have been modified via the cache. **_DCacheSyncRegion** differs from **_DCacheSync** in that the latter call affects the entire cache, whereas **_DCacheSyncRegion** may affect only a portion of it.

The obsolete name **_syncmem** is a synonym for **_DCacheSyncRegion**.

8.3 _DCacheFlush

`#include <nuon/cache.h>`
void **_DCacheFlush**(*void*)

Synchronizes memory with the MPE's data cache and invalidates the data cache. After this call is made all data is guaranteed to have been stored out to memory, and all data cache tags have been marked as invalid. This function is usually used before trying to access memory that may have been modified by another MPE. It is a more expensive operation than **_DCacheSync**, and should therefore be used sparingly.

The obsolete name **_flushcache** is a synonym for **_DCacheFlush**.

8.4 _DCacheInvalidateRegion

`#include <nuon/cache.h>`
void **_DCacheInvalidateRegion**(void * *startaddr*, void * *endaddr*)

Marks any data cache lines containing addresses from *startaddr* up to (but not including) *endaddr* as invalid. This means that any dirty data in those cache lines will be lost. It is *strongly* recommended that both *startaddr* and *endaddr* be on cache line boundaries (the default data cache line size is 32 bytes).

This call may be useful for synchronizing memory transfers between MPEs, but it must be used with extreme caution. **_DCacheInvalidateRegion** does not check for dirty data in the cache, so it should be used only for blocks of memory which are only read by this MPE (not written), are cache line aligned, and are a multiple of the cache line size in length.

8.5 _CacheConfig

`#include <nuon/cache.h>`
void **_CacheConfig**(unsigned int *dcachectl*, unsigned int *icachectl*)

Changes the cache configuration of the BIOS MPE. *dcachectl* is the requested new value for the data cache control register, and *icachectl* is the requested new value for the instruction cache control value. If either of these is `-1`, then the corresponding cache control register will be left unchanged. Otherwise, these values may be built using the following macros:

CACHE_DIRECT	direct mapped cache
CACHE_2WAY	2 way associative cache
CACHE_3WAY	3 way associative cache
CACHE_WAYSIZE_1K	each way is 1024 bytes
CACHE_WAYSIZE_2K	each way is 2048 bytes
CACHE_WAYSIZE_4K	each way is 4096 bytes
CACHE_BLOCKSIZE_16	each cache line is 16 bytes
CACHE_BLOCKSIZE_32	each cache line is 32 bytes
CACHE_BLOCKSIZE_64	each cache line is 64 bytes
CACHE_BLOCKSIZE_128	each cache line is 128 bytes

The data cache value must leave at least 1024 bytes of memory for BIOS use (so on a 4K MPE at most 3K may be used for the data cache). The amount of memory used for the cache is the product of the number of ways and the size of each way. For example, a 3 way cache with 1K waysize uses 3K of memory.

All of instruction memory may be used for instruction cache; the BIOS does not require any instruction memory to be set aside. However, note that the assembler and compiler are set up to produce code with a cache line size of at least 32 bytes; to run code with a smaller line size requires special configuration (and is also likely to produce poor results; in general larger line sizes are better for code).

The C library **read** and **write** functions may not work properly if the data cache line size is greater than 32 bytes. The **_MediaRead** function also assumes a cache line size of 32 bytes. So in practice the data cache line size must be either 16 or 32 bytes.

Calling **_CacheConfig** has the side effect of flushing the data cache (as with a **_DCacheFlush** call), as well as invalidating the contents of the instruction cache. This happens even if one or both parameters is `-1`. Calling **_CacheConfig** with *icachectl* set to `-1` is the only legal way to flush the instruction cache. This may be necessary before running code loaded from disc (depending on how the code was loaded).

9. Audio Functions

The audio BIOS provides calls to initialize and set up the audio hardware. Because the actual hardware (type and number of DACs) can vary on different products using NUON technology, libraries use these BIOS calls to make sure application may run on all platforms. It is unlikely that application programmers will need to make any of these calls directly; instead they should use the appropriate audio library (such as libnise) to set up audio and play sounds.

9.1 _AudioReset

`#include <nuon/audio.h>`
void **_AudioReset**(*void*)
Resets the audio hardware. Same code that is used in the boot-up sequence.

9.2 _AudioQuerySampleRates

`#include <nuon/audio.h>`
long **_AudioQuerySampleRates**(*void*)
This function returns a bitfield of the supported sample rates

Bit	Rate	define
0	44.1 kHz	`RATE_44_1_KHZ`
1	88.2 kHz	`RATE_88_2_KHZ`
2	22.05 kHz	`RATE_22_05_KHZ`
4	48 kHz	`RATE_48_KHZ`
5	96 kHz	`RATE_96_KHZ`
6	24 kHz	`RATE_24_KHZ`
8	32 kHz	`RATE_32_KHZ`
9	64 kHz	`RATE_64_KHZ`
10	16 kHz	`RATE_16_KHZ`

All other bits are reserved.

9.3 _AudioSetSampleRates

`#include <nuon/audio.h>`
void **_AudioSetSampleRates**(long *rateField*)
With this call you set one of the allowed sample rates. If you set an unsupported rate, the call will be ignored. If you try to set more than one bit in the bitfield, only the most significant set bit is tested. This call also initializes the S/PDIF output.

9.4 _AudioQueryChannelMode

`#include <nuon/audio.h>`
long **_AudioQueryChannelMode**(*void*)

This function returns the current selected channel mode, as follows:

Stream Mode	
Two 16bit channels	`STREAM_TWO_16_BIT`
Four 16bit channels	`STREAM_FOUR_16_BIT`
Two 32bit channels	`STREAM_TWO_132_BIT`
Eight 16bit channels	`STREAM_EIGHT_16_BIT`
Eight 32bit channels	`STREAM_EIGHT_32_BIT`
Four 32bit channels	`STREAM_FOUR_32_BIT`
Audio DMA buffer size	
1K	`BUFFER_SIZE_1K`
2K	`BUFFER_SIZE_2K`
4K	`BUFFER_SIZE_4K`
8K	`BUFFER_SIZE_8K`
16K	`BUFFER_SIZE_18K`
32K	`BUFFER_SIZE_32K`
64K	`BUFFER_SIZE_64K`
Audio control	
Enable Audio DMA	`ENABLE_AUDIO_DMA`
Enable wrap interrupt	`ENABLE_WRAP_INT`
Enable half interrupt	`ENABLE_HALF_INT`
Enable sample interrupt	`ENABLE_SAMP_INT`
Enable DMA skip mode	`ENABLE_DMA_SKIP`
Enable DMA stall mode	`ENABLE_DMA_STALL`

The channel mode information may be extracted from the returned value by methods TBD.

9.5 _AudioSetChannelMode

`#include <nuon/audio.h>`
void **_AudioSetChannelMode**(long *mode*)

The parameter *mode* specifies a channel mode by combining one stream mode, one DMA buffer size, and any number of audio control defines. These defines are described in the documentation for the **_AudioQueryChannelMode** function.

9.6 _AudioSetDMABuffer

`#include <nuon/audio.h>`
void **_AudioSetDMABuffer**(long *dmaBaseAddr*)

dmaBaseAddr specifies the base address for audio DMA, *i.e.* the address of the audio data buffer.

10. Video Functions

10.1 Video Data Structures

10.1.1 VidDisplay

The `VidDisplay` structure has the following fields:

```
/*
 * structure for overall display configuration
 */
typedef struct bios_viddisplay {
    int dispwidth;
    int dispheight;
    int bordcolor;
    int progressive;
    int fps;
    short pixel_aspect_x;
    short pixel_aspect_y;
    short screen_aspect_x;
    short screen_aspect_y;
    int reserved[3];
} VidDisplay;
```

The fields have the following meanings:

`dispwidth` The width of the physical display (use `-1` to request a default width). Note that this is the actual number of pixels output to the CCIR656 output stream, and on almost all systems will be a constant which cannot actually be modified.

`dispheight` The height of the display in pixels (use `-1` to request a default width).

`bordcolor` The border color (as a 32bpp color). The **_VidSetBorderColor** function may be used to modify this.

`progressive` A flag for interlaced (if 0) or progressive (if 1) display. Use `-1` for the default value. This must be set to 0 or to `-1` in the current BIOS version.

`fps` Fields per second (frames per second for a progressive display). This value is a 16.16 fixed point number. On virtually all systems this field is read only. `fps` should be set to 0 (which means “use default value”) in the structure passed to the **_VidConfig** function. Note that most older systems return 0 in this field, so unfortunately it is not very useful.

`reserved` This space is reserved for future expansion. It *must* be set to 0.

In the present version of the BIOS, `dispwidth` and `dispheight` must be set to the default values (either `-1` or the values returned by **_VidQueryConfig**), and `progressive` must be set to 0.

10.1.2 VidChannel

The `VidChannel` structure holds information about a specific video channel (main channel or overlay channel). The structure has the following elements:

```
/* structure for configuring a specific channel */
typedef struct bios_vidchannel {
    long dmaflags;
    void *base;
    int  dest_xoff;
    int  dest_yoff;
    int  dest_width;
    int  dest_height;
    int  src_xoff;
    int  src_yoff;
    int  src_width;
    int  src_height;
    unsigned char clut_select;
    unsigned char alpha;
    unsigned char vfilter;
    unsigned char hfilter;
    int reserved[5];
} VidChannel;
```

The fields have the following meanings:

`dmaflags` DMA flags for reading from the bitmap to be displayed on the channel. These are normally for a pixel mode transfer. The main channel also supports MPEG-2 video images. To set up an MPEG video image in the main channel, set the DMA flags as follows: bits 16-23 are the width of the frame buffer in macroblocks (in other words, the width divided by 16); bits 14-15 must be set to 2 (for MCU transfers); bits 4-7 must be set to 0; and bit 0 is 0 for progressive source material and 1 for interlaced source material. Note that not all BIOSes support interlaced source material properly.

`base` The base address in memory (SDRAM) of the bitmap to be displayed on the channel. For MPEG images (supported on the main channel only) this must be the address of the luma data, with the chroma data immediately following it in memory.

`dest_xoff` The horizontal offset for the screen image, in pixels. If this is `-1`, then the offset will be calculated automatically to center the image horizontally.

`dest_yoff` The vertical offset for the screen image, in pixels. If this is `-1`, then the offset will be calculated automatically to center the image vertically.

`dest_width` The width of the channel on screen, in pixels.

`dest_height` The height of the channel on screen, in pixels.

`src_xoff` The horizontal offset into the source bitmap for the displayed image. Only a portion of the source bitmap needs to be displayed. This field is an integer giving the offset in pixels.

`src_yoff` The vertical offset into the source bitmap for the displayed image. Only a portion of the source bitmap needs to be displayed. This field is an integer giving the offset in pixels.

`src_width` The width of the source image to be displayed. Starting from `src_xoff`, this many pixels will be fetched and will be scaled up (or down) to fit into `dest_width` pixels on screen. This field is an integer giving the width in pixels.

Note that the overlay channel only supports 1-1 and 2-1 scaling, therefore `src_width` must be either the same as `dest_width`, or be one half of it.

`src_height` The height of the source image to be displayed. Starting at `src_yoff`, this many lines of pixels will be fetched and will be scaled up (or down) to fit into `dest_height` lines on screen. This field is an integer giving the height in pixels.

`clut_select` This field is used only when a 4 bit per pixel bitmap is being displayed, and selects which 16 entries from the 256 entry CLUT to use for the display. `clut_select` gives the index of the first of the 16 CLUT entries to use.

`alpha` This field is used only when a 16 bit per pixel bitmap is being displayed, and gives the alpha value to use at each pixel. This has an effect only for the overlay channel; for the main channel, this value must always be set to 0. Note that 4, 8, and 32 bit per pixel bitmaps already contain an alpha value (either explicitly, or implicitly in the CLUT) for each pixel.

`vfilter` The vertical filter to use when displaying this channel on screen.

`hfilter` The horizontal filter to use when displaying this channel on screen. This field is currently ignored by the BIOS; the hardware has a fixed horizontal filter for the main channel, and no filter for the overlay channel.

`reserved` This space is reserved for future expansion, and must be filled with zeros.

10.2 Video Format Restrictions

In the present version of the BIOS, the following restrictions exist:

1. Only the main channel may display MPEG pixels.
2. For MPEG format output (supported on the main channel only) the `src_xoff` and `src_yoff` fields must be 0, and `src_width` and `src_height` fields must be the entire MPEG frame buffer. The `base` structure must point at the luma data for the frame buffer, and the chroma data must immediately follow the luma data in memory.

3. Only the OSD channel may have 4 or 8bpp displays; hence, the `clut_select` field is meaningful only for the OSD channel.

4. For an 8bpp CLUT based channel, `src_width` and `src_xoff` must both be even. For a 4bpp CLUT based channel, they must both be multiples of 4.

5. On the OSD channel, the destination width must either be equal to the source width, or be twice the source width.

6. On the main channel, the destination width and source width are independent (so arbitrary scaling ratios may be achieved). However, beware that shrinking an image (*i.e.* having `dest_width` less than `src_width` place a demand on main bus bandwidth proportional to the ratio of the two widths. In practice with current hardware it is not practical to shrink more than 2:1, so `src_width` should not be more than twice as big as `dest_width`.

7. For both the main and OSD channels, the source and destination heights must be even (a multiple of 2).

8. For both the main and OSD channels, the source width and initial source `x` offset must be multiples of the DMA quantum (see the table below).

9. It is the caller's responsibility to ensure that the various `xoff`, `yoff`, `width`, and `height` fields are set properly.

10. The `hfilter` field must be set to `VID_HFILTER_4TAP` for the main channel, and to `VID_HFILTER_NONE` for the overlay channel.

11. The `vfilter` flag may be one of `VID_VFILTER_NONE`, `VID_VFILTER_2TAP`, or `VID_VFILTER_4TAP`. However, please note that filtering a CLUT based image is highly unlikely to work as desired, so for all practical purposes if the OSD channel is set to display a CLUT image then the `vfilter` flag for it must be set to `VID_VFILTER_NONE`.

10.2.1 Frame Buffer Widths and DMA Quanta

The hardware restricts frame buffer widths to certain sizes, based on the number of bits per pixel (BPP) and whether or not the `DMA_CLUSTER_BIT` is set in the DMA flags for the frame buffer. In the table below, *quantum* is the value which the width must be a multiple of, and *max* is the maximum width.

	CLUSTER=0		CLUSTER=1	
BPP	*quantum*	*max*	*quantum*	*max*
4	32	4096	64	8192
8	16	2048	32	4096
16	8	1024	16	2048
32	8	512	8	1024

For example, a 16 bit per pixel frame buffer with `DMA_CLUSTER_BIT` clear must be a multiple of 8 pixels wide, and may be at most 1024 pixels wide.

Note that when displaying a subwindow of the frame buffer, the *x* offset of the subwindow and the width of the subwindow must both be multiples of the DMA quantum.

For MPEG format displays, the width must be a multiple of 16 pixels and may be at most 1024 pixels wide. It is not possible to display a subwindow of an MPEG frame buffer; the entire frame buffer must be displayed.

10.2.2 Frame Buffer Heights

Ordinary pixel frame buffers must be a multiple of 8 pixels high if `DMA_CLUSTER_BIT` is clear, and a multiple of 16 if it is set. It is possible to display a subwindow of a pixel mode frame buffer, for example the first 198 lines out of 200.

MPEG frame buffers must always be a multiple of 16 pixels high, and the entire frame buffer must be used as a source.

10.3 _VidConfig

`#include <nuon/video.h>`
int **_VidConfig**(VidDisplay * *display*, VidChannel * *mainchannel*, VidChannel * *osdchannel*, void * *reserved*)

Configures the video display.

display is a pointer to a `VidDisplay` structure that specifies the overall video display configuration. If it is `NULL`, then the overall video configuration remains unchanged (so the last configuration set by **_VidConfig** is used; the boot code calls **_VidConfig** once to initialize the display, so there is always a valid configuration to use). Please note that on most machines it is not possible to modify the `dispwidth`, `dispheight`, and `progressive` fields of the display structure; on most systems these properties are always determined by the video mode (PAL or NTSC) and cannot be changed.

mainchannel is a pointer to a `VidChannel` structure giving the configuration for the main display channel. *osdchannel* is a pointer to a `VidChannel` structure giving the configuration for the on-screen display (OSD) channel (a channel overlaid on top of the main channel). If either is `NULL`, then the corresponding channel will not be displayed.

_VidConfig returns 1 on success, and 0 on failure. It will fail (and return 0) if any reserved field is non-zero, or if an illegal value is requested for any parameter.

10.4 _VidQueryConfig

`#include <nuon/video.h>`
int **_VidQueryConfig**(VidDisplay * *display*)

Fills in the fields of the `VidDisplay` structure with the current values as set by the last call to **_VidConfig**. Returns an integer specifying what the current video display system is; this is 1 for NTSC and 2 for PAL.

10.5 _VidSetup

void **_VidSetup**(void * *base*, long *dmaflags*, int *width*, int *height*, int *vfilter*)

This is a simplified interface to the **_VidConfig** function. It sets up the main video channel to display a source image that is *width* by *height* pixels large, the base of which (in SDRAM) is at address *base*. *dmaflags* are the DMA flags used to refer to the source image (for either reading or writing; **_VidSetup** can accept either). The overlay channel is not enabled.

The output image is full screen, and has a vertical filter as specified by *vfilter*. See the description of **_VidConfig** for which vertical filters are permitted on the main channel.

10.6 _VidChangeBase

`#include <nuon/video.h>`
int **_VidChangeBase**(int *which*, long *dmaflags*, void * *base*)

Changes the video display base for either the main or OSD video channel.

which determines the channel that is to be modified, either `VID_CHANNEL_MAIN` or `VID_CHANNEL_OSD`.

dmaflags specifies the flags for reading from a channel; flags for writing to the frame buffer may also be used here, since **_VidChangeBase** will always set the read bit. In the current BIOS release, the *dmaflags* must always be for a pixel mode DMA.

base specifies the base address of the frame buffer.

The video display will not actually be modified until the next vertical blanking interval. The **_VidSync** function may be used to wait for this.

_VidChangeBase returns 1 on success, 0 if the specified channel is not currently active or if any other error is detected.

10.7 _VidChangeScroll

`#include <nuon/video.h>`
int **_VidChangeScroll**(int *which*, int *xoff*, int *yoff*)

Changes the X and Y offsets into the source frame buffer. The changes do not take place immediately; instead, they are queued and will be executed at the time of the next vertical blanking interval. If two or more calls to **_VidChangeScroll** are made in the same field, only the last call will take effect.

which determines the channel that is to be modified, either `VID_CHANNEL_MAIN` or `VID_CHANNEL_OSD`.

xoff and *yoff* are integers giving the pixel offset of the new source origin. It is the caller's responsibility to ensure that *xoff* and *yoff* are valid.

_VidChangeScroll returns 0 if the specified channel is not currently active.

10.8 _VidSync

`#include <nuon/video.h>`
long **_VidSync**(int *n*)

Wait for some number of vertical blanking intervals.

If *n* is a positive integer, then the **_VidSync** function waits for that many fields to pass before returning. Each field occupies 1/60 of a second for NTSC, and 1/50 of a second for PAL video output.

If *n* is `0`, and a vertical blanking interval has occurred since the last call to **_VidConfig**, **_VidChangeBase**, or **_VidChangeScroll**, then **_VidSync** will return immediately; otherwise it will wait for one vertical blanking interval to occur. When it returns, **_VidSync** returns the (new) value of the BIOS internal field counter giving the number of fields elapsed since the last reset.

If *n* is `-1`, then **_VidSync** will not wait at all, but will immediately return the current value of the BIOS internal field counter.

If *n* is `-2`, the **_VidSync** will wait until a time when it would be convenient to hand video over to the presentation engine. The exact video line will depend on the BIOS and PE versions. This internal functionality of **_VidSync** is intended for the DVD player and similar system applications, and should not be used by games.

10.9 _VidSetBorderColor

`#include <nuon/video.h>`
void **_VidSetBorderColor**(unsigned int *color*)

Set the (32 bit) border color to *color*. The border color is displayed wherever there is active video but neither the main channel nor overlay channel are active.

10.10 _VidSetCLUTRange

`#include <nuon/video.h>`
void **_VidSetCLUTRange**(int *i*, int *num*, unsigned long *colors[]*)

Update a set of consecutive entries in the VDG color look up table with the specified colors. *i* is the index of the first CLUT entry to update. *num* is the number of consecutive entries to fill. *colors* is an array of *num* 32 bit colors (in YCrCbAlpha format) to use to set CLUT entries *i* through *i+num*-1 inclusive.

The VDG CLUT is used for 4 and 8 bit per pixel display modes, in which the pixel value is an index into the CLUT. These modes are only allowed in the overlay channel; the main channel has no CLUT mode. The CLUT is not used in 16 and 32 bit per pixel display modes.

10.11 _VidSetOutputType

`#include <nuon/video.h>`
void **_VidSetOutputType**(int *type*)

Select video output type. If *type* is `kVidOutputComposite`, the video output will be optimized for composite video; if it is `kVidOutputSvideo`, it will be optimized for S-Video output. If it is `kVidOutputDefault`, an appropriate value will be chosen based on the settings provided by the user.

This interface is provided for the DVD player and similar system applications. It is very unlikely that “normal” applications like games will ever need to make this call. At the time a game is started, the video output will already have been initialized by the firmware to be the value selected by the user as optimal.

11. Controllers and Other Serial Devices

The NUON BIOS supports up to 8 joysticks plus one infrared controller. All controllers present a common interface, which may be extended for specialized controllers (such as steering wheels or "6 degrees of freedom" joysticks).

The BIOS continually monitors the state of all controllers connected to the system, and puts the data associated with them into the global array pointed to by _*Controller*. This array has 9 entries. Entry 0 is for the infrared remote (if present). The other entries are for joysticks plugged into the two controller ports on the console. Each port could have up to 4 controllers plugged into it through the use of a multi-port adaptor. Therefore, entries 1–4 are for "Port 1", and entries 5-8 are for "Port 2". If no multi-port adaptors are in use, then only entries 1 and 5 could actually have joysticks plugged in; thus a "typical" configuration might have player 1 using _*Controller[1]* and player 2 using _*Controller[5]*.

11.1 Performance

Reading a large number of joysticks (particularly joysticks with analog pots) can impose significant overhead on the system, since it requires that high speed serial bus packets be sent to each joystick. There are two ways to reduce the impact on applications of the joystick processing. One is for the application to call **_ControllerPollRate** to set the polling rate for the serial bus to a lower value. (This option is practical if the application is not using a modem or other high speed device connected to the bus.)

Another way to improve performance is to clear all of the analog bits in the `properties` fields of any controllers for which analog joystick data is not actually being used by the application. The analog bits are all of the properties bits except `CTRLR_STDBUTTONS`, `CTRLR_DPAD`, `CTRLR_SHOULDER`, and `CTRLR_EXTBUTTONS`.

After all analog related property bits are cleared, the BIOS will no longer attempt to read the corresponding analog parts of the corresponding joystick. Restoring the bits will again enable the analog device to work. If the application wishes to do this, it must save the property bits before clearing any of them.

Note that if the joystick is unplugged and then plugged in again all of the property bits (including the analog ones) will be reset, and will have to be cleared again.

11.2 Data Structures

Each controller entry has the following format:

```
#include <nuon/joystick.h>

typedef volatile struct
{
// scalar 0
     unsigned int      changed : 1;
     unsigned int      status : 1;
     unsigned long     manufacture_id : 8;
     unsigned long     properties : 22;
 // scalar 1
     unsigned short    buttons;

     union
     {
          signed char     xAxis;
          signed char     wheel;
          signed char     paddle;
          signed char     rodX;
     } d1;

     union
     {
          signed char     yAxis;
          signed char     rodY;
     } d2;

// scalar 2
     union
     {
          signed char      xAxis2;
          unsigned char    throttle;
          unsigned char    throttle_brake;
     } d3;

     union
     {
          unsigned char    yAxis2;
          unsigned char    brake;
          signed char      rudder;
          signed char      twist;
     } d4;

     union
     {
          signed char     quadjoyX;
          signed char     mouseX;
          signed char     thumbwheel1;
          signed char     spinner1;
          signed char     reelY;
     } d5;

     union
     {
          signed char     quadjoyY;
          signed char     mouseY;
          signed char     thumbwheel2;
          signed char     spinner2;
          signed char     shuttle;
     } d6;
 // scalar 3
     unsigned long      remote_buttons;
} ControllerData;

extern ControllerData *_Controller;
```

The meaning of the fields is as follows:

`changed` If this is 0, then the `status` has not changed. If it is 1, then the `status` has changed (that is, the controller has been plugged in or unplugged) or else the controller itself has changed in some way which may require an update of its device driver since the last time the controller was read by the BIOS.

Thus, for example, if a new joystick is plugged in during play of a game, the BIOS will set both the `changed` and `status` bits of the corresponding entry in the *_Controller*. When the game notices that the `changed` bit is set, it should call **_DeviceDetect** to configure the new joystick (or other device). **_DeviceDetect** will clear the `changed` bit.

Note that while `changed` is set, the data in the rest of the structure may not be valid. For this reason it is very important to call **_DeviceDetect** when the change is noticed.

`status` If this is 0, then the joystick is not connected and no data is valid. If it is 1, and `changed` is 0, then the joystick is functioning properly and the data in the rest of the structure is correct.

`manufacturer_id` This field is reserved for manufacturers, and may be used to give further information about the controller, for example to indicate that some special extended information is available.

`properties` This field indicates information about the specific controller, and about how to interpret the remaining fields in this structure. The following bits are defined:

Symbolic Name	Meaning
CTRLR_STDBUTTONS	Standard buttons: "A", "B", "Start", and "Select", are available in `buttons`.
CTRLR_DPAD	8 way (4 switch) directional D-pad buttons are available in the `buttons` field.
CTRLR_SHOULDER	Left and right shoulder triggers are active in the `buttons` field.
CTRLR_EXTBUTTONS	For all controllers except CTRLR_MOUSE, indicates extended standard buttons: "C", "D", "E", and "F" are present in the `buttons` field. For CTRLR_MOUSE, indicates the presence of a third center button mapped to "C" in the `buttons` field.
CTRLR_ANALOG1	Primary analog stick; if the CTRLR_QUADJOY1 bit is also set, indicates (to the BIOS) that the controller is really a Quadrature device, and that the BIOS should cook the data to look like an analog device. The joystick position may be read from the `d1.xAxis` and `d2.yAxis` fields.
CTRLR_ANALOG2	Secondary analog stick, may be read from the `d3.xAxis` and `d4.yAxis` fields.
CTRLR_QUADJOY1	Quadrature joystick device. The quadrature motion values are available in `d5.quadjoyX` and `d6.quadjoyY` fields. If the CTRLR_ANALOG1 bit is also set, the BIOS will cook the data to look like an analog device and place the results into `d1.xAxis` and `d2.yAxis`. Applications should check CTRLR_ANALOG1 and if it is set use the analog values.
CTRLR_WHEEL	Analog steering wheel – "A" and "B" buttons are also assumed present. Additional buttons may be indicated by the appropriate flag bits. Data for the wheel is in `d1.wheel`.
CTRLR_PADDLE	Paddle controller. "A" and "B" buttons are assumed present in `buttons`, and the paddle position is in `d1.paddle`.
CTRLR_THROTTLEBRAKE	Accelerator / Throttle and brake returned together in same value (in the `d3.throttle_brake` field). Negative values indicate brake, positive values indicate throttle.
CTRLR_THROTTLE	Accelerator / Throttle, value in `d3.throttle`.
CTRLR_BRAKE	Brake; value is in `d4.brake`.
CTRLR_RUDDER	Rudder; value is in `d4.rudder`.

Symbolic Name	Meaning
`CTRLR_TWIST`	Analog twist control – This would indicate a joystick with a analog twist axis. Usable as a flight rudder. The value is in the `d4.twist` field.
`CTRLR_MOUSE`	Mouse device; has buttons "A" (left button), "B" (right button), and optionally (if `CTRLR_EXTBUTTONS` is set) a "C" button (middle button) in the `buttons` field. Quadrature motion information returned in the `d5.mouseX` and `d6.mouseY` fields.
`CTRLR_TRACKBALL`	Trackball device; otherwise similar to `CTRLR_MOUSE` (*q.v.*).
`CTRLR_QUADSPINNER1`	A quadrature spinner wheel like the Tempest arcade controller or an old Atari 2600 driving controller. Only two buttons ("A" and "B") are expected to be available, but if `CTRLR_EXTBUTTONS` is set then a third "C" button is also available. The quadrature motion information is returned in the `d5.spinner1` field. Positive movement values indicate clockwise rotation.
`CTRLR_QUADSPINNER2`	A quadrature spinner wheel with motion information available in the `d6.spinner2` field. Otherwise this is like `CTRLR_QUADSPINNER1` (*q.v.*).
`CTRLR_THUMBWHEEL1`	A quadrature thumbwheel. This is usually combined with other attributes. If no other attributes are selected, then only two buttons ("A" and "B") are available, unless `CTRLR_EXTBUTTONS` is also set, in which case a third ("C") button is also available. Quadrature motion information is returned in the `d5.thumbwheel1` field.
`CTRLR_THUMBWHEEL2`	A quadrature thumbwheel with quadrature motion information available in `d6.thumbwheel2`. Otherwise similar to `CTRLR_THUMBWHEEL1`.
`CTRLR_FISHINGREEL`	A fishing reel controller. "A" and "B" buttons are assumed present. Accelerometer information for the rod is returned in the `d1.rodX` and `d2.rodY` fields. Quadrature motion information for the reel is returned in the `d5.reelY` field.
`CTRLR_REMOTE`	Infrared (or RF) remote controller. Buttons are in the `buttons` and `remote_buttons` fields.
`CTRLR_EXTENDED`	Device supports extended information calls, and may have additional capabilities such as force feedback or downloadable information.

Some of the bits in the `properties` field are writable; see joystick performance (section 11.1) for how to disable the polling of analog devices.

`buttons` This field indicates which of the standard joystick buttons are currently pressed. Bits that are set to 1 are pressed; bits set to 0 are not. The following buttons are defined as "standard joystick buttons":

Symbolic Name	Meaning
`JOY_A`	Primary fire button (right thumb)
`JOY_B`	Secondary fire button (right thumb)
`JOY_C`	Right thumb button 3
`JOY_D`	Right thumb button 4
`JOY_E`	Right thumb button 5
`JOY_F`	Right thumb button 6
`JOY_START`	Start/Play button
`JOY_RIGHT`	D-pad right
`JOY_UP`	D-pad up
`JOY_LEFT`	D-pad left
`JOY_DOWN`	D-pad down
`JOY_R`	right shoulder button
`JOY_L`	left shoulder button
`JOY_Z`	trigger button

`remote_buttons` Additional buttons. The interpretation of these buttons depends on the controller. For infrared remote (IR) controllers there is a standard set of buttons defined, as follows:

Symbolic Name	Meaning
`IR_STOP`	stop button
`IR_SETUP`	setup button
`IR_ENTER`	enter button
`IR_RESUME`	resume button
`IR_DISPLAY`	display button
`IR_MENU`	menu button
`IR_TOP`	top menu button
`IR_ANGLE`	angle button
`IR_SUBTITLE`	subtitle button
`IR_AUDIO`	audio control button
`IR_ZOOM`	zoom button (may also set `d2`)
`IR_VOLUME`	volume button (usually also sets `d2`)
`IR_SKIP_NEXT`	skip to next chapter
`IR_SKIP_PREV`	skip to previous chapter
`IR_FF`	fast forward
`IR_FR`	fast reverse
`IR_SF`	slow forward
`IR_SR`	slow reverse
`IR_CLEAR`	clear button
`IR_KEY_0` ... `IR_KEY_9`	numeric keys
`IR_EJECT`	eject button

Please note that some keys found on an infrared remote will correspond to joystick keys found in the `buttons` field (for example, the "enter" key will typically map to `JOY_A` and the "play" key to `JOY_START`).

Also note that some remotes may have a "shuttle" or similar multi-position control device. The driver for such remotes will cause some appropriate button bits to be set, and put a parameter into `d6.shuttle` to indicate a value for the shuttle. For example, it might set `IR_FF` with a value in `d6.shuttle` of 32 to indicate a very fast forward.

Similarly, the `IR_VOLUME` and `IR_ZOOM` bits will generally be accompanied by values in `d6.shuttle` to indicate the desired change in volume and/or zoom.

The entries in *_Controller* may be read by the application at any time.

11.3 Implementation

The *_Controller* variable is actually a pointer which is initialized in the C run time startup code (`crt0.o`) by a call to **_ControllerInitialize**. This is transparent to the user application. In the unlikely event that your application is entirely written in assembler and does not link with `crt0.o`, then you will need to declare the variable *_Controller* and initialize it with the value returned by **_ControllerInitialize** (or make some equivalent arrangements).

11.4 _ControllerInitialize

`#include <nuon/bios.h>`
ControllerData * **_ControllerInitialize**(*void*)

Initializes the controller system and returns a pointer to the BIOS *_Controller* array. This function is called automatically by the C run time initialization code, so it need not be called by user applications.

11.5 _ControllerExtendedInfo

`#include <nuon/joystick.h>`
void * **_ControllerExtendedInfo**(int *slot*)

Returns a pointer to any extra information associated with the serial device (usually a controller) which is in slot *slot*. This extra info is device dependent.

_ControllerExtendedInfo will return 0 if no extra information is available (which will be the usual case).

11.6 _ControllerPollRate

`#include <nuon/joystick.h>`
void **_ControllerPollRate**(int *n*)

This function will set the rate at which the NUON will poll the joystick bus for status. The number n *i*s in milliseconds in the range 0 to 1023, and represents one less than the number of milliseconds between poll attempts. Thus, a parameter of

0 means that the joystick is polled every millisecond, and 4 means that it is polled every 5 milliseconds. The actual poll rate is determined by the system clock, which on existing systems has a 200 Hz granularity. Thus, sensible values for n *a*re 0, 5, 10, 15, and so on. The larger the poll rate interval, the fewer resources are used by the joystick serial bus, but the lower the rate which characters may be received. High speed modems require a poll rate of 0 (the default); ordinary joysticks perform well at poll rates of 15 or even higher.

11.7 _DeviceDetect

`#include <nuon/bios.h>`
int **_DeviceDetect**(int *slot*)

Whenever a new device is plugged in to the serial device chains, the BIOS will set the `changed` bit in the corresponding *_Controller* array slot. When the application notices that this bit has been set, it should call **_DeviceDetect** to determine the exact characteristics of the device and to configure it for proper operation. After the call to **_DeviceDetect** the `changed` bit will be cleared, and the `status` and `properties` bits in the *_Controller* array element will be properly configured for whatever device(s) are currently occupying that slot in the array.

slot is the index in the *_Controller* array of the device that has changed.

_DeviceDetect returns `-1` if in fact `_Controller[slot]` has not changed, and otherwise returns the new value of the `status` bit for that controller slot.

Please note that devices other than controllers may occupy positions on the serial device chain. Such devices will have a `status` value of `0`, since there is no controller in that position.

11.8 _BiosIRMask

`#include <nuon/bios.h>`
unsigned long **_BiosIRMask**(int *mode*, unsigned long *mask*)

Controls which keys on the IR remote are handled automatically by the BIOS, and which will be passed to applications. On some NUON players, keys like “Power” and “Eject” may not take effect automatically, but rather send events to the NUON system; on other players, these keys are handled automatically by other parts of the firmware (e.g. front panel micros).

On systems for which the keys send events, the BIOS normally handles the events transparently to the application. It is possible (but usually unwise) to override this handling, and **_BiosIRMask** provides this capability. Most applications should use the built in BIOS handling, since this ensures a uniformity of look and feel across platforms and applications.

mode is 0 to report the current mask value (the keys for which the BIOS handles events automatically) and 1 to change this value. *mask* is a bitmap of the remote buttons which the application wants the BIOS to handle automatically. This is only used if *mode* is 1, and should typically be formed from a previous return value from **_BiosIRMask**. The return value is the new mask which the BIOS will use. The bit mask is constructed using the same values as returned in the `remote_buttons` field of the CONTROLLERS structure

If the application requests the BIOS to handle keys which it is not prepared to handle, it will ignore those bits in the mask and set them to 0 in the returned value.

12. Time Related Functions

12.1 _TimeOfDay

`#include <nuon/time.h>`
int **_TimeOfDay**(struct _currenttime * *tm*, int *getset*)

This function gets or sets the current time of day according to the "wall clock". If the parameter *getset* is `0`, then the structure pointed to by *tm* is filled in with the current time. This structure contains (at least) the following fields:

`sec` Seconds past the minute, in the range `0` – `59`.

`min` Minutes past the hour, in the range `0` – `59`.

`hour` Hours since midnight, in the range `0` – `23`.

`wday` Day of the week, in the range `0` (Sunday) – `6` (Saturday).

`mday` Day of the month, in the range `1` – `31`.

`month` Month, in the range `1` (January) – `12` (December).

`year` Year, in the range `2000` – `9999`. (The BIOS is not guaranteed to be Y10K compliant.)

`isdst` A flag indicating whether or not Daylight Savings Time is in effect. The value is `0` if DST is not in effect, positive if it is, and `-1` if the hardware clock does not keep this information.

`timezone` The current time zone, expressed as the minutes west of the Greenwich meridian (negative for time zones east of Greenwich). Thus, for example, Eastern Standard Time, which is 5 hours west of Greenwich, would be stored as 300 (5*60). Note also that the `timezone` field always stores the standard time zone; to represent Eastern Daylight Savings Time, `timezone` still contains 300, but `isdst` is set to `1`.

If *getset* is `1`, then the hardware clock is set from the structure pointed to by *tm*.

_TimeOfDay returns `0` if successful, or `-1` if there is an error. For example, if there is no accessible hardware clock present on this machine and the software clock has not yet been initialized, any **_TimeOfDay** calls to retrieve the time will return `-1` until the time has been initialized by a "set" call to **_TimeOfDay**.

12.2 _TimeElapsed

`#include <nuon/time.h>`
unsigned long **_TimeElapsed**(long * *secs*, long * *usecs*)

Determine the time elapsed since the system was last reset. The number of seconds elapsed is written into the long word pointed to by *secs*, if this is not `NULL`. The number of microseconds elapsed within the second is written into the long word pointed to by *usecs*, if this is also not `NULL`. Please note that the BIOS timer is not guaranteed to be microsecond accurate; on most platforms it will however be more than millisecond accurate (but see below for bugs on some early BIOS versions).

In any event, **_TimeElapsed** returns the number of milliseconds since boot. If the *usecs* parameter is `NULL` then the milliseconds returned will be accurate only to the system timer resolution (so it may be off by up to 4 milliseconds).

Note that a bug in some early versions of the BIOS may cause **_TimeElapsed** to fail to return a proper count of microseconds when *usecs* is not `NULL`. When this bug is triggered, the count of microseconds may actually be negative, and will certainly be wrong; and the milliseconds returned by **_TimeElapsed** will also be wrong. A patch to work around this will be incorporated into the SDK, but will result in *usecs* being accurate only to system timer resolution (which is 5 milliseconds).

12.3 _TimerInit

`#include <nuon/time.h>`
int **_TimerInit**(int *n*, int *rate*)

Initializes one of the hardware timers. *n* specifies which timer to initialize. *rate* specifies the number of microseconds between interrupts. After this call, the timer interrupt for timer *n* will be triggered every *rate* microseconds. It is up to the application to catch this interrupt with the **_IntSetVector** call and perform the appropriate actions.

_TimerInit will return 1 on success, and 0 if an attempt is made to initialize a reserved timer (namely timer 0, which is reserved for BIOS use) or a timer that does not exist.

12.4 _TimeToSleep

`#include <nuon/time.h>`
void **_TimeToSleep**(long *msecs*)

Waits for *msecs* milliseconds to pass.

13. Media and Drive Functions

The goal of the media related functions in the BIOS is to allow DVD and set-top applications to access their data in similar ways, despite many differences in the data rates and storage capabilities of local storage and network storages. It also allows us to provide emulation services, running on PC's, for debugging of applications.

In general, applications will be encouraged to place all of their data in one "file", which will be a physical file on a DVD or CD, or an abstract network connection in a set top box environment. This will help to hide the differences between systems, and will also allow for efficient use of and access to local media (where available). Avoiding the use of a file system (except for the initial open) will allow the BIOS to make use of various media-specific optimizations; for example, to avoid disk seeks or pause/restart sequences.

13.1 _MediaOpen

`#include <nuon/mediaio.h>`

int **_MediaOpen**(int *device*, const char * *name*, int *mode*, int * *blocksize*)

Initializes and opens a device for access by the media library. *device* specifies the type of device to be opened. This may be one of the constants given below:

`MEDIA_BOOT_DEVICE`	The device from which the application was loaded.
`MEDIA_DVD`	The local DVD player, if it exists.
`MEDIA_CABLE`	The cable network, if it exists.
`MEDIA_EMULATOR`	The PC debug emulator.
`MEDIA_NVRAM`	Non-volatile memory.

name is a pointer to a string that gives further information for file system access on the device, if applicable. Normally this will be either `NULL` or an empty string, indicating that the default file for the system should be used. This parameter would be used, for example, to open a specific file on the PC when using the debug emulator.

mode describes how the application wishes to use the opened device. In the present version of the BIOS, this should always be `0`; future uses of *mode* are TBD.

blocksize is a pointer to an integer, which is filled in with the size of blocks on this device.

Returns a non-zero handle on success, or a `0` on failure.

13.2 _MediaClose

`#include <nuon/mediaio.h>`

int **_MediaClose**(int *handle*)

Closes a handle previously returned by **_MediaOpen**.

13.3 _MediaGetDevicesAvailable

`#include <nuon/mediaio.h>`

long **_MediaGetDevicesAvailable**(*void*)

Returns a bitmapped set of flags indicating which media devices are available on this system. See the **_MediaOpen** function for details on which devices might be supported.

Most programs will not need this function, simply using `MEDIA_BOOT_DEVICE` as the device for media I/O.

13.4 _MediaGetInfo

`#include <nuon/mediaio.h>`
int **_MediaGetInfo**(int *handle*, struct MediaInfo * *info*) Gets information about the media device whose handle is *handle*. *info* is a pointer to a `MediaInfo` structure. This structure contains the following fields:

`sectorsize` An integer giving the size of blocks on the device, in bytes. Typically this will be 2048 bytes, but some devices (such as networks, hard disks, and particularly flash memories) may have different block sizes.

`device` The device corresponding to the handle. This is the value originally passed to **_MediaOpen** when the handle was first created.

`datarate` An estimate of the number of bytes that can be read from the device per second. This may or may not be accurate, particularly for network-based devices where the actual throughput will vary depending on network load.

_MediaGetInfo returns 0 if it is successful, and an error code if not (for example, if *handle* is invalid).

13.5 _MediaRead

`#include <nuon/mediaio.h>`
int **_MediaRead**(int *handle*, int *mode*, long *startblock*, long *blockcount*, void * *buffer*, void *(*callback)(int status, long blocknum)*)

Reads sectors from the current device. *handle* is a handle returned by a previous **_MediaOpen** call. The *mode* parameter specifies how callbacks are to take place. If this is `MCB_END` then the callback will be made only at the end of the transfer or if there is an error. If it is `MCB_EVERY` then the callback is made for every block transfered. *startblock* and *blockcount* give the starting block for the transfer and the number of blocks to read, respectively. The block size is returned by the **_MediaOpen** function, and may also be obtained via **_MediaGetInfo**. *buffer* is a pointer to memory (in either SDRAM or system RAM) into which the data will be written. If the memory is to be accessed via the data cache, see the important note below.

The *callback* function is called either at the end of the transfer or after every block. It is also called if an error is detected. Its main use is to allow the application to keep track of what has been read (for example, to show a progress bar). The parameters to *callback* are:

status This will be the *mode* parameter originally passed to **_MediaRead**, if operation is proceeding normally, or else `MCB_ERROR` if an error has occurred.

blocknum The number (relative to the start of the read) of the last block that was successfully read before the callback, or `-1` if no sectors were successfully read. This is useful mainly when the callback is happening after every block (*mode* is `MCB_EVERY`), or to detect errors.

The **_spinwait** function is provided as a simple callback function.

_MediaRead returns a `0` on success, or `-1` if it fails (for example, if there is no room in the queue for pending reads).

Important: The **_MediaRead** function may not necessarily update the data cache correctly with the data that has just been read. For this reason, the *buffer* parameter *must* be aligned on a data cache line boundary (32 bytes). Also, the data cache for the buffer region should be marked as invalid after the **_MediaRead** has completed. This may be done with the **_DCacheInvalidateRegion** call.

13.6 _MediaWrite

`#include <nuon/mediaio.h>`
int **_MediaWrite**(int *handle*, int *mode*, long *startblock*, long *blockcount*, void * *buffer*, void *(*callback)(long status, long blocknum)*)

Writes sectors to the current device. This call is similar to the **_MediaRead** function, except that the data is written to the device rather than read from it. If the device is read-only, the **_MediaWrite** will return `-1` when first called.

13.7 _spinwait

`#include <nuon/mediaio.h>`
long **_spinwait**(long *status*, long *blocknum*)

A simple callback function for **_MediaRead** or **_MediaWrite**. **_spinwait** sets the global variable *_MediaWaiting* to 0 when it is called with *status* indicating a successful read, and to a negative error number when it is called with *status* indicating an error. **_spinwait** has no purpose other than as a callback for **_MediaRead** or **_MediaWrite**.

Please note that the BIOS uses **_spinwait** internally in the implementation of file system access functions and in the UDF filesystem read code. Do not use **_spinwait** yourself if you make use of any of these services.

13.8 _MediaShutdownMPE

`#include <nuon/mediaio.h>`
void **_MediaShutdownMPE**(*void*)

The BIOS must use an auxiliary MPE to run the DVD disk drive. Typically this is done by allocating the top 4K of both instruction and data memory on MPE 0, and running the drive management code there. Applications are free to use the bottom

4K, with some restrictions (TBD). However, there may be times when the application needs to use all of MPE 0, and does not need to make any media calls. In such situations, the application may make a **_MediaShutdownMPE** call to free as many MPE resources as possible. After the **_MediaShutdownMPE** call, no media calls (including indirect calls caused by the streaming audio library) may be made until after a successful **_MediaInitMPE** call has been made.

PLEASE NOTE: **_MediaShutdownMPE** has the side effect of stopping the media MPE completely and marking it as free. Applications should be sure that any code they have running on the media MPE (for example because they have called **_MPEAlloc** with the `MPE_HAS_MINI_BIOS` flag) has finished before they call **_MediaShutdownMPE**.

13.9 _MediaInitMPE

`#include <nuon/mediaio.h>`
int **_MediaInitMPE**(*void*)

Initialize the BIOS drive handling code running on another MPE after a call to **_MediaShutdownMPE** (q.v.). **_MediaInitMPE** will attempt to allocate an MPE for the BIOS to use to run drive code. It will prefer to allocate "half" of MPE 0 (leaving some room for user code there). Future BIOS versions will try other MPEs if MPE 0 is not available. If no MPE could be allocated, then **_MediaInitMPE** returns `-1`; in this case the application must free any MPEs it owns and try again. When **_MediaInitMPE** succeeds, it returns the number of the MPE which BIOS code is running on. This MPE is marked allocated by the BIOS, so the application does not need to do anything special with the returned value.

13.10 _DiskGetTotalSlots

`#include <nuon/mediaio.h>`
int **_DiskGetTotalSlots**(*void*)

This returns the number of disk slots available on this system. This will be 0 on set top boxes with no drive attached, 1 on typical DVD players, and more than 1 for DVD players with a carousel.

13.11 _DiskChange

`#include <nuon/mediaio.h>`
int **_DiskChange**(int *flags*, int *DestSlot*, unsigned int * *NewSlot*)

Attempt to move the disk carousel to a new disk slot, or determine the current disk slot.

flags controls how the other parameters are interpreted. If this is `SLOT_IS_DELTA`, then *DestSlot* is a signed delta to the current slot; if it is `SLOT_IS_ABS` then it is an absolute slot number (where 1 is the first slot). If *DestSlot* is 0, then the carousel will not be moved; do this to inquire what the current slot is.

NewSlot, if nonzero, will point to a 32 bit integer where the new disk slot number (or current disk slot number, if *DestSlot* is 0) will be placed. This number is 1 based and is valid only if the return value from **_DiskChange** is non-negative.

If *DestSlot* is 0, **_DiskChange** will always return 0. Otherwise, **_DiskChange** will return one of the following values:

0 The carousel was moved to the desired slot, and it contains a DVD with a valid `NUON.DAT` file.

1 The carousel was moved to the desired slot, it contains a DVD, but no `NUON.DAT` file was found.

2 The carousel was moved to the desired slot, but it contains media other than a DVD (*e.g.* an audio CD).

3 The carousel was moved to the desired slot, and that slot is empty.

4 The carousel was not moved, because the requested slot is the same as the current slot.

-1 *flags* was invalid.

-2 Invalid (out of range) *DestSlot*.

-3 Unable to determine current slot; carousel not moved.

13.12 _DiskEject

`#include <nuon/mediaio.h>`
int **_DiskEject**(*void*)

Issue a command to eject the disk tray. This function will only work on DVD players and similar devices which actually have a disk. Note that the call may return before the tray is actually open. In practice this should cause no problems: retracting the tray again (with **_DiskRetract**) should be done only after some signal from the user that the disk has been changed.

_DiskEject returns `0` on success (if the command to open the tray has successfully been queued). It returns `-1` if there is no disk device present.

13.13 _DiskRetract

`#include <nuon/mediaio.h>`
int **_DiskRetract**(*void*)

Issue a command to retract the disk tray. This function will only work on DVD players and similar devices which actually have a disk. Note that the call may return before the tray is fully retracted. Any calls which attempt to access the drive while the tray is still retracting will timeout and will have to be retried. It is a good idea to wait for a second or two after the **_DiskRetract** call has returned before attempting such calls, since on some platforms it may take longer than this to recover from the failed media calls.

_**DiskRetract** returns `0` on success (if the command to close the tray has successfully been queued). It returns `-1` if there is no disk device present.

14. File and Directory Operations

14.1 Introduction

The file and directory operations described in this chapter form a higher level interface to disk and device operations. They sit on top of the functions described in the Media and Drive Functions (section 13) chapter, and as such, are less efficient. The Media functions should be used where high performance is required (e.g. in games). The file and directory functions are most useful in porting applications from other platforms, or where performance is not an issue.

14.1.1 File and Directory Names

All file names (including directories) are ASCII strings. File names may consist of (English) letters (either upper or lower case), digits, or the hyphen (`-`) character. Many file systems will allow for other ASCII characters as well. Whether case is significant or not will depend on the individual file system; for the UDF and ISO9660 file systems (the ones most commonly used) case is not significant, but for the remote file system used in development systems it may be (*i.e.* the files `FOO` and `Foo` may be different). It is best to make a consistent use of case in your file names and to not assume anything about whether case is significant.

Forward slashes (`/`) are used to separate the directory names in a path. The first "directory" name in a full path is actually the name of the file system or device. For example, to read the file `NUON.RUN` from the directory `NUON` on a DVD (with the UDF file system), the full path/file name is `/udf/NUON/NUON.RUN`.

14.2 Standard C Functions

The usual C standard library functions from the `stdio.h` header are available on NUON. File names are as described in File and Directory Names (section 14.1.1). In addition, many functions from the POSIX standard are available in the NUON C libraries. For example, **open**, **read**, **close**, and **lseek** are present with full functionality. The **stat** and **lstat** functions are also available, but do not work properly on all file systems. Always be sure to check the return values from any C library call.

The POSIX **opendir** and **readdir** functions are not currently available. The BIOS **_FindName** function may be used to obtain similar functionality.

14.3 _FindName

`#include <nuon/bios.h>`
int **_FindName**(const char * *path*, int *which*, char * *buf*, int *buflen*, int * *err*)

Finds a particular file name in a directory. *path* is the full path of the directory. *which* is a (0 based) integer giving which file or directory name within the *path* is to be returned. *buf* points to an area of memory in which the file or directory name will be written. *buflen* is the length (in bytes) of the *buf* area. *err* points to an integer into which any error will be placed.

If **_FindName** succeeds, it returns 0 and fills *buf* with the file name. If the name actually refers to a subdirectory, then it will have a forward slash (`/`) appended.

If it would succeed, but the length *buflen* of the buffer *buf* is too small to hold the file name. If *which* is more than the number of entries in the directory *path*, then **_FindName** returns 0, and sets *buf* to an empty string. Otherwise (on an error), **_FindName** returns `-1` and sets *err* to a particular error code.

The **_FindName** function is most often used to iterate over the names in a directory. For example, to get a list of all files and subdirectories of a directory *dir*, you could do:

```
void
list(const char *dir)
{
  int i;
  int n;
  int err;
  char buf[BUFSIZ+1]; // leave space for trailing 0

  for (i = 0; ;i++) {
    n = _FindName(dir, i, buf, BUFSIZ, &err);
    if (n < 0) {
      printf("_FindName returned error %d\n", n);
      break;
    } else if (n > 0) {
      printf("directory entry %d too long: skipped\n", i);
    } else {
      printf("%s\n", buf);
    }
  }
}
```

To list subdirectories recursively, there are two options. One is to use the POSIX **stat** to look at the attributes of each file in the directory. A simpler solution is to check for the forward slash appended by **_FindName** to directory names.

15. Memory Management

15.1 Local MPE Memory

The BIOS must use some local memory for certain functions (such as interrupt service routines). The memory left over may be used by applications and libraries. The BIOS maintains a pool of "scratch space" which libraries and applications may use. This scratch space is not guaranteed to be preserved across library or (some) BIOS function calls, so it should be used only in "leaf" functions or in between function calls.

The BIOS DMA, comm bus, and time functions are guaranteed not to touch the local scratch memory. Other BIOS calls (including media calls) might modify the scratch memory, so it is not safe to assume that it is preserved across other BIOS calls.

15.2 _MemLocalScratch

`#include <nuon/bios.h>`
void * **_MemLocalScratch**(unsigned int * *size*)

Finds the size and location of "scratch" memory in local data RAM to be used by libraries and by applications. *size*, if non-zero, points to an integer which will be filled in with the number of bytes of local memory available. The return value is a pointer to the first byte of the scratch memory. This memory is vector aligned, and at least 512 bytes long.

15.3 SDRAM and System RAM

15.4 _MemAlloc

`#include <nuon/bios.h>`
void * **_MemAlloc**(unsigned long *size*, unsigned *align*, unsigned *flags*)

Allocates memory from SDRAM (or, in some limited circumstances, from system RAM). *size* is the amount of memory to allocate (in bytes). *align* is a memory alignment requirement; for example, to allocate on a long word boundary, set *align* to 4. If *align* is 0, then the default (vector) alignment is used. *flags* is used to control how the allocation is done. If *flags* is `kMemSDRAM`, then SDRAM is allocated. If it is `kMemSysRam`, then system RAM is allocated. *flags* may also be set to the logical or of those two values, in which case **_MemAlloc** first tries to allocate from system RAM, and if that fails tries SDRAM.

_MemAlloc returns a pointer to the newly allocated memory, or 0 if the allocation failed.

Please note that in the default C run time environment the C compiler has already allocated all of the system RAM for the **malloc** pool, so the C function **malloc** (or its relatives) must be used to allocate memory from system RAM. **_MemAlloc** may be used only to allocate memory from SDRAM; in other words, the *flags* parameter should always be `kMemSDRAM`. The `kMemSysRam` parameter to **_MemAlloc** is only useful to device drivers and similar internal BIOS applications.

Also note that the BIOS does not flag portions of SDRAM allocated in the applications COFF file. Thus, **_MemAlloc** may return memory that the application is already using! To avoid this bug, place any SDRAM sections in the COFF file at the top of SDRAM (up to `0x407fffff`).

15.5 _MemFree

`#include <nuon/bios.h>`
void **_MemFree**(void * *ptr*)

Frees a block of memory previously allocated by **_MemAlloc**. *ptr* is the pointer returned by **_MemAlloc**.

15.6 _MemAdd

`#include <nuon/bios.h>`
void **_MemAdd**(unsigned long *baseaddr*, unsigned long *size*, unsigned *flags*)

This functions adds a block of memory to the BIOS memory allocator. It should not, generally, be called by applications. It is provided for the use of device drivers for memory cards whose memory is directly accessible by MPE 3. *baseaddr* is the base address at which the memory may be accessed (typically this will be on the other bus). *size* is the size of the memory. *flags* is the kind of memory, either `kMemSysRam` for system RAM, or `kMemSDRAM` for SDRAM.

15.7 _MemInit

This is an internal BIOS function to initialize the memory system. It should not be called by applications.

16. Platform Control Functions

These functions serve as a basis for certain somewhat machine-specific settings that are generally used to provide customization features to hardware manufacturers. They are not appropriate for general use, and are not necessarily implemented on any particular system. In the event that a function documented here is not supported, it will return -1 and have no other effect.

16.1 _StartImageValid

```
#include <nuon/bios.h>
```
int **_StartImageValid**(*void*)

If the last attempt to write a new MPEG startup image was completed successfully, this function will return 1. If not, it will return zero. If the current system does not support an MPEG startup image, it will return -1.

16.2 _SetStartImage

```
#include <nuon/bios.h>
```
int **_SetStartImage**(void * *ptr*)

Copies a 720x576 pixel MPEG image into a reserved block of flash ROM for display during system boot-up. Returns 0 on success; 1 on (detected) failure, and -1 if unsupported.

16.3 _GetStartImage

```
#include <nuon/bios.h>
```
void * **_GetStartImage**(*void*)

Returns the address of the stored MPEG startup image, or -1 if unimplemented.

17. Inside the BIOS

This chapter describes the internal organization of the BIOS. It attempts to provide a rationale for the design of the BIOS, in order to aid future maintainers.

17.1 Memory Layout

The BIOS uses several sections of the RAM memory map, plus the system ROM or flash. Within MPE 0, it uses (together with the audio libraries) the upper 4K of each of iram and dtram to support reading from a disk and playing streaming audio. In MPE 3, it uses 256 bytes at address `0x20300c00` in iram and 256 bytes at address `0x20100c00` in dtram for DMA routines, cache invalidation code, and a few other things. The 64K starting at `0x80000000` (in system RAM) is used for jump tables, BIOS internal data structures, and a few user-visible data structures. The area beginning at `0x80760000` contains all other ram-resident BIOS code and data, as well as a memory arena for downloaded device drivers and other required memory allocation.

The Presentation Engine, when activated, uses the memory between address `0x804e0000` and the beginning of BIOS memory. The Presentation Engine is present and active at application boot-time, unless it is overwritten during application loading. In that event, it can be re-loaded and initialized by calling **_InitPE**.

17.2 Bootup and Initialization

The BIOS is responsible for initializing the NUON chip, the board that the chip is on, and itself. After initialization is complete, the BIOS loads and starts the bootup application that is stored in the system ROM.

17.3 Devices

The various input/output devices that can be connected to a NUON system are all abstracted through the device layer. This permits software to interact with devices that were not yet designed when the software was written. Device drivers can be either resident, meaning compiled into the BIOS when it was built, or downloaded from the device itself and installed.

The three device driver types are File drivers, Media drivers, and Network drivers. Any particular device will require a driver of at least one of the types. Some devices will need all three. For example, the host PC for a development system can act as a device providing all three types of services.

The fundamental operation of all of these is the same, but the particular services offered varies with the type. All the installed drivers of a particular type are held in a linked list. When a new instance of use of a device is initiated, the appropriate chain is searched for a driver that can (or will) handle the request. That driver is then associated with that use. When subsequent calls are made on that particular use, the associated driver is called at the correct entrypoint for the requested service. In the case of media devices, there is only one active device permitted at any particular time, though formerly active devices may still be servicing queued requests. For the other device types, each instance generates an entry in the system descriptor table, called `filelist`. Later calls reference it via a handle.

17.3.1 File Drivers

The file driver structure is defined in the file `"syscalls/fs.h"`. File drivers provide support for the C library file manipulating functions **close**, **fstat**, **ioctl**, **isatty**, **link**, **lseek**, **lstat**, **open**, **read**, **stat**, **unlink**, and **write**.

17.3.2 Media Drivers

The media driver structure is defined in the file `"syscalls/media.h"`. Media drivers provide support for the BIOS media functions described in section 13.

17.3.3 Network Drivers

The network driver structure is defined in the file `"syscalls/fs.h"`. Network drivers provide support for the library networking functions **accept**, **bind**, **close**, **connect**, **gethostname**, **getpeername**, **getsockname**, **getsockopt**, **ioctl**, **listen**, **read**, **recv**, **recvfrom**, **recvmsg**, **send**, **sendmsg**, **sendto**, **sethostname**, **setsockopt**, **shutdown**, **socket**, and **write**.

17.4 Library Support

The bios provides all necessary support for the Cygnus “newlib” C library, either directly or through device drivers.

17.5 Debugging

Most debugging code within the BIOS consists of assertions which are enabled on development systems. Typically, if an assertion fails the chip is halted via the `halt` instruction. At this point, the `rx` register should contain a pointer to an ASCII string explaining the reason for the halt.

18. Release List & Known Bugs

18.1 Samsung N2000 (Extiva-1)

The model-specific executable extension for the Samsung N2000 Extiva-1 is "`exv`".

18.1.1 Bios version 1.00.31

CVS tag `version1_00_31`, built 13 June, 2000.

This version was used in approximately the first 1000 units manufactured.

The **_NetAccept** function is defective. A patch is in the standard C library `accept()` function, and in bios releases starting with version 1.00.32.

There is a bug in the miniBIOS DMA functions. These functions fail to wait for the main bus DMA pending flag to go clear before beginning a transfer. As of this writing, there is no BIOS patch available. The bug is fixed in version 1.00.34 and later.

There is a set of problems with memory devices on the Nuon Peripheral Bus. These are fixed in version 1.00.33 and later. No patch/workaround is presently available.

1. Raw reads/writes were conditionalized out. Now they are exported unconditionally — and renamed to what would make sense.
2. An unformatted (or damaged) card was unaccessible, and could hang the system. Made them available for raw access.
3. The BIOS relied on the card's magic number, which effectively excluded an unformatted card. Added a test to check for the presence of a card.
4. The card's bus coordinates were not available to the application.

18.1.2 Bios version 1.00.34

CVS tag `version1_00_34`, built 21 June, 2000.

No bugs known.

This build has fixes for the **_NetAccept**, miniBIOS and memory device problems mentioned above, and also calls **_BiosReboot** instead of **_BiosExit** in several circumstances. It also handles some additional keys from the remote control.

18.1.3 Bios version 1.00.37

CVS tag `version1_00_37`, built 17 August, 2000.

No bugs known.

This includes some changes in the handling of remote buttons, and also a timing change in the NUON port handling.

18.1.4 Bios version 1.00.41

CVS tag `version1_00_41`, built 3 October, 2000.

No bugs known.

This is nearly unchanged from version 1.00.37. The differences involve some new authentication keys that can be used optionally, controlled by configuration.

18.2 Toshiba SD2300

The model-specific executable extension for the Toshiba SD-2300 is "`sd2300`".

18.2.1 Bios version 1.02.52

CVS tag `NUON0052`.

The miniBIOS contains a bug which can cause it to sometimes not notice missing bytes from the CDI. This can cause **_MediaRead** (and related functions, including the C **read** function) to sometimes get bad data. A patch to the miniBIOS code was introduced in the C startup code `crt0.o` as of SDK release 0.82.

The **_TimeElapsed** system call will return improper values for both microseconds and milliseconds elapsed if the *usec* parameter is not `NULL` (in other words, if microsecond accurate timing is requested). To work around this, always pass `NULL` for the second parameter of **_TimeElapsed**. A patch for this problem will be introduced in SDK release 0.83, but this patch will cause all times (including microseconds) to be accurate only to the nearest 5 milliseconds.

18.2.2 Bios version 1.02.57

CVS tag `NUON0057ENG1`.

This version was mistakenly released by Toshiba. It was never approved for production by either VM Labs or Toshiba. There was apparently a labelling blunder in the factory. It went into SD-2300 serial numbers 0XCM200001–0XCM203370, 0XCM400313–0XCM404630, 0XDM100505–0XCM102000, 9CM402163–09CM402215, and 09CM402185–09CM402208.

The last two ranges of serial numbers are probably wrong.

This version has a bug in the authentication code that can cause a crash when attempting to load an application (and possibly a driver) from a NUON bus memory device.

The miniBIOS bug described above also affects this version.

The **_TimeElapsed** bug described above also affects this version.

18.2.3 Bios version 1.02.59

CVS tag `NUON0059`.

The miniBIOS contains a bug which can cause it to sometimes not notice missing bytes from the CDI. This can cause **_MediaRead** (and related functions, including the C **read** function) to sometimes get bad data. A patch to the miniBIOS code was introduced in the C startup code `crt0.o` as of SDK release 0.82.

The **_TimeElapsed** system call functions incorrectly when called with a request for microsecond accuracy (the second parameter is not `NULL`); this is the same bug which affects Bios version 1.02.52.

18.2.4 Bios version 1.02.61

CVS tag `NUON0061`.

This contains a fixed miniBIOS which correctly flags errors when bytes go missing from the CDI.

The **_TimeElapsed** system call functions incorrectly when called with a request for microsecond accuracy (the second parameter is not `NULL`); this is the same bug which affects Bios version 1.02.52.

18.3 Samsung N501 (Extiva-2)

The model-specific executable extension for the Samsung N2000 Extiva-2 is “`n501`”.

18.3.1 Bios version 1.03.40

CVS tag `Extiva2-1_3_0040`

The **_VidQueryConfig** system call fills in only the first 36 bytes of the 40 byte structure, due to an error in the header files used to build this BIOS. The unfilled 4 bytes is reserved for future use, so this doesn’t actually cause any harm.

Index