VM LABS

Nuon Miscellaneous Utility Functions

Programmers' Manual

April 11, 2001

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

Copyright notice

This is a preliminary specification. VM Labs reserves the right to make changes to any and all of the interfaces described in this document.

Contents

1 Introduction **1**

2 Functions **2**

2.1 Comm Bus Functions . . . 2
2.1.1 CommSend . . . 2
2.1.2 CommSendInfo . . . 2
2.1.3 CommRecv . . . 2
2.1.4 CommRecvInfo . . . 3
2.1.5 CommRecvQuery . . . 3
2.1.6 CommRecvQueryInfo . . . 3
2.1.7 CommSendRecv . . . 3
2.1.8 _comm_send . . . 4
2.1.9 _comm_recv . . . 4
2.1.10 _comm_recv_query . . . 4
2.2 Timer Functions . . . 4
2.2.1 GetTimer . . . 5
2.3 Video Functions . . . 5
2.3.1 VidSetup . . . 5
2.4 MPE Control Functions . . . 5
2.4.1 ReadMPERegister . . . 5
2.4.2 WriteMPERegister . . . 6
2.4.3 StopMPE . . . 6
2.4.4 WaitMPE . . . 6
2.4.5 StartMPE . . . 6
2.4.6 CopyToMPE . . . 7
2.5 DMA Functions . . . 7
2.5.1 _mpedmaregister . . . 7
2.5.2 _raw_plotpixel . . . 8
2.5.3 _synccache . . . 8
2.5.4 _flushcache . . . 8
2.6 Fixed Point Math Functions . . . 9
2.6.1 DoubleToFix . . . 9
2.6.2 FixToDouble . . . 9
2.6.3 FixMul . . . 9
2.6.4 FixDiv . . . 9
2.6.5 FixRecip . . . 9
2.6.6 FixSinCos . . . 10
2.6.7 FixSqrt . . . 10
2.6.8 FixRSqrt . . . 10
2.7 SDRAM memory functions . . . 10
2.7.1 SDRAMAlloc . . . 10
2.7.2 SDRAMFree . . . 10
2.7.3 SDRAMInit . . . 11

2.8 Miscellaneous Functions 11
2.8.1 _GetLocalVar 11
2.8.2 _SetLocalVar 11
2.8.3 msprintf 11
2.8.4 DebugWS 12

1. Introduction

The `libmutil` library collects together a number of useful utility functions for the NUON architecture. All of these functions are callable from C, and follow the C calling conventions, which are summarized here:

1. The C compiler automatically prefixes an underscore to all function and variable names. So, for example, the function documented here by its C name **CommSend** would be called **_CommSend** from assembly language.

2. The first ten arguments are passed in registers `r0` through `r9`. Other arguments are passed on the C stack (pointed to by `r31`).

3. The C stack pointer is general purpose register `r31`. The hardware stack pointer `sp` is reserved for interrupt purposes. Note that the C stack grows downwards (predecremented), and must always be vector aligned. Some of the `libmutil` functions use the hardware stack pointer `sp`, so it must be properly initialized before any of these functions are called. The standard C startup code will do this automatically.

4. A function's return value is placed in `r0`. 64 bit values are returned in `r0` and `r1`. Functions which return vector values return them in `v0`.

5. C functions may modify general purpose registers `r0` through `r11` (*i.e.* vector registers `v0`, `v1`, and `v2`), and also register `r29`. The other general purpose registers (`r12` through `r28`, `r30`, and `r31`) are preserved.

6. I/O registers concerned with bus transfers (the main bus DMA registers, other bus DMA registers, and comm bus registers) may be modified if appropriate to the function (obviously a function like **CommSend** will modify the comm bus!). All other I/O registers are preserved, including (in particular) the `sp` and `acshift` registers.

7. `acshift` must not be modified – it must always be left set to 0, unless interrupts are turned off.

8. All functions were assembled to use the cache, in its default configuration (32 byte lines). If you wish to use 16 byte cache lines, you will have to recompile the library from its source. (The library should be compatibile as is with cache lines larger than 32 bytes.)

2. Functions

2.1 Comm Bus Functions

The communication bus functions listed below are simple interfaces to the BIOS comm bus functions, and are provided primarily for backwards compatibility. See the BIOS documentation for further details of their operation.

2.1.1 CommSend

`#include <nuon/mutil.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.

NOTE: beware of using **CommSend** followed by **CommRecv** to retrieve register data from hardware if data may arrive unexpectedly from another source. It is probably better to use **CommSendRecv** to query hardware registers via the comm bus.

2.1.2 CommSendInfo

`#include <nuon/mutil.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 which is 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, *info* is ignored.

2.1.3 CommRecv

`#include <nuon/mutil.h>`
int **CommRecv**(**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*. **CommRecv** returns the comm bus id of the processor which sent the packet.

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

NOTE: there are a number of tricky synchronization issues involved in using the comm bus, since many interrupt routines must use it to program the hardware or to read joystick data. If you wish to use the comm bus to read hardware registers, you should use the **CommSendRecv** function to ensure an atomic send and reply.

2.1.4 CommRecvInfo

`#include <nuon/mutil.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.* 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 which 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 (which returns immediately if no data is available) use **CommRecvQueryInfo**.

2.1.5 CommRecvQuery

`#include <nuon/mutil.h>`
int **CommRecvQuery**(**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*. **CommRecvQuery** returns the comm bus id of the processor which sent the packet. If no packet is available at the time of this call, **CommRecvQuery** returns immediately with a return value of -1, and the memory pointed to by *packet* is left unchanged.

For a blocking read function (which waits until a packet is received) use **CommRecv**.

2.1.6 CommRecvQueryInfo

`#include <nuon/mutil.h>`
int **CommRecvQueryInfo**(**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 bits of extra information will be placed in the memory pointed to by *info* (assuming, of course, that extra information was sent along with the packet; otherwise it will contain garbage).

CommRecvQueryInfo returns the comm bus id of the processor which sent the packet. If no packet is available at the time of this call, **CommRecvQueryInfo** returns immediately with a return value of -1, and the memory pointed to by *packet* and *info* is left unchanged.

For a blocking read function (which waits until a packet is received) use **CommRecvInfo**.

2.1.7 CommSendRecv

`#include <nuon/mutil.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 which was received.

CommSendRecv locks 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 which are accessible only via the comm bus.

2.1.8 _comm_send

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

Sends a communication bus packet to the destination whose communication bus id is *target*. The packet consists of the four long words *p0*, *p1*, *p2*, and *p3*. An additional 8 bits of information may be sent to targets which are MPEs; the low order 8 bits of *info* contains this extra information. This function is otherwise similar to **CommSendInfo** and has the same limitations.

2.1.9 _comm_recv

```
#include <nuon/mutil.h>
```
long **_comm_recv**(*void*)

Receives a single communication bus packet, and returns the first word of that packet. This is not terribly useful in C, but this function is very useful when called from an assembly language program, since in fact registers `r0` to `r3` are set to the packet contents, `r4` is set to the comm bus id of the sender, and `r5` is set to the received extra comm bus information (if any). C programmers will find the **CommRecvInfo** function to be more useful. **_comm_recv** will block until a comm bus packet has been received, if necessary.

2.1.10 _comm_recv_query

```
#include <nuon/mutil.h>
```
long **_comm_recv_query**(*void*)

Receives a single communication bus packet, and returns the first word of that packet. This is not terribly useful in C, but this function is very useful when called from an assembly language program, since in fact registers `r0` to `r3` are set to the packet contents, `r4` is set to the comm bus id of the sender, and `r5` to the received extra comm bus information (if any).

C programmers will find the **CommRecvQueryInfo** function to be more useful to them. The **_comm_recv_query** function will return immediately if no comm bus packet is available to be read, with `r4` set to -1.

2.2 Timer Functions

The timer functions are simply interfaces to the BIOS timer routines, and are kept in the utility library for backwards compatibility only. It is suggested that new applica-

tions call the BIOS routines directly.

2.2.1 GetTimer

```
#include <nuon/mutil.h>
```
long **GetTimer**(**long ***_secs_,**long ***_usecs_)

Gets the elapsed time since the BIOS was last initialized. If *secs* is nonzero, then the number of seconds elapsed is placed in the long word pointed to by it. If *usecs* is nonzero, then the number of microseconds since the last second is placed in the long word pointed to by *usecs*. This time is probably accurate only to tens of microseconds (don't rely on the least significant digit).

GetTimer returns the number of milliseconds since the BIOS was last initialized, which is probably sufficient resolution for most needs. Note that this will wrap around after approximately 1 month.

GetTimer is identical to the BIOS **_TimeElapsed** function.

2.3 Video Functions

The utility library video functions provide a simplified interface to the BIOS video functions. See the BIOS manual for a description of the more comprehensive BIOS video routines.

2.3.1 VidSetup

```
#include <nuon/mutil.h>
```
void **VidSetup**(**void ***_baseaddr_, **long** _dmaflags_, **int** _width_, **int** _height_, **int** _filter_)

Initializes the main video channel to display a frame buffer. *baseaddr* is the address of the frame buffer; *width* and *height* are its width and height, respectively. *dmaflags* are the DMA flags used to access the memory for writing; only the pixel type and cluster bit fields of this are actually used. *filter* specifies what kind of vertical filtering is to be used for the video output: 0 or 1 means no filter, 2 means a 2 tap filter, and 4 means an (expensive) 4 tap filter.

The given frame buffer will be displayed full screen on the video; in other words, it will be scaled up to 720 by 480 pixels (for NTSC; 576 pixels for PAL).

2.4 MPE Control Functions

2.4.1 ReadMPERegister

```
#include <nuon/mutil.h>
```
long **ReadMPERegister**(**int** _mpe_, **void ***_regaddr_)

Reads the value of a register in another MPE. *mpe* is the number of the MPE from which to read. *regaddr* is the address of the register to read, expressed as a "standard" (relative to MPE 0) address. Returns the value of the register.

NOTE: reading registers `r0` – `r31` of a running MPE may cause that MPE to crash.

2.4.2 WriteMPERegister

```
#include <nuon/mutil.h>
```
void **WriteMPERegister**(**int** *mpe*, **void ***regaddr*, **long** *value*)

Writes a new value to an MPE's register. *mpe* is the number of the MPE to which the value is to be written. *regaddr* is the address of the register to write, expressed as a "standard" (relative to MPE 0) address. *value* is the new value to write into the register.

NOTE: writing registers `r0` – `r31` of a running MPE may cause that MPE to crash.

2.4.3 StopMPE

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

Stops an MPE. *mpe* is the number of the MPE to stop. If the MPE is already stopped, this function does nothing.

2.4.4 WaitMPE

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

Waits for an MPE to stop. *mpe* is the number of the MPE to wait for. If the MPE is already stopped, this function returns immediately. Otherwise, it blocks until the MPE stops (for example by executing a `halt` instruction).

2.4.5 StartMPE

```
#include <nuon/mutil.h>
```
void **StartMPE**(**int** *mpe*, **void ***codestart*, **long** *codesize*, **void ***datastart*, **long** *datasize*)

Loads both code and data into an MPE, and then causes that MPE to start executing at address `0x20300000`. If the MPE is already running at the time **StartMPE** is called, it will be stopped first and then restarted with the new code and data. *mpe* is the number of the MPE affected. It must *not* be the ID of the currently running MPE. *codestart* is the address of the code to be loaded into the MPE's instruction memory, starting at address `0x20300000`. *codesize* is the size of the code in bytes; this must be smaller than the size of the MPE's instruction memory (typically 4096 bytes) and must be a multiple of 4. *datastart* is the address of the data to be loaded into the MPE's data memory, starting at address `0x20100000`. *datasize* is the size of the data in bytes; this must be smaller than the size of the MPE's data memory (typically 4096 bytes).

The MPE will start with its `pcfetch` register pointing to `0x20300000` (the base of instruction RAM) and its stack pointer `sp` pointing to the address `0x20101000`

(usually the end of data RAM). The `rz` register will be set to `0`. The MPE should stop itself by issuing a `halt` instruction.

Note: On the beta (Oz) hardware we recommend that *codesize* and *datasize* be made multiples of 16, and that *codestart* and *datastart* should start on vector boundaries; this will ensure that the other bus page boundary bug is avoided. If the code and data are placed in unique sections (for example, `foocode` and `foodata`) then the linker will ensure that these conditions are met. Placing all the code (or data) for an invocation of **StartMPE** in its own segment also makes it easy to find the *codestart* and *codesize* values, since for each segment *foo* the linker will create symbols *foo_start* and *foo_size*. Note that the linker created symbols are absolute symbols, and must therefore be referred to as though they were addresses.

For example: if an external assembly language file creates the segments `mycode` and `mydata` for some function, this function can be loaded into MPE 2 and executed via:

```
extern int mycode_start[], mycode_size[];
extern int mydata_start[], mydata_size[];

StartMPE(2, mycode_start, (long)mycode_size,
            mydata_start, (long)mydata_size);
```

2.4.6 CopyToMPE

`#include <nuon/mutil.h>`
void **CopyToMPE**(**int** *mpe*, **void ***dest*, **void ***src*, **long** *size*)

Copies *size* bytes of data from *src* to *dest*, which is memory inside MPE number *mpe*. *dest* must be an MPE-relative address; *src* is a system (absolute) address.

WARNING: this function may not work properly if the destination MPE is running.

void **CopyFromMPE**(**int** *mpe*, **void ***dest*, **void ***src*, **long** *size*)

Copies *size* bytes of data from *src*, which must be an MPE-relative address for memory inside MPE *mpe*, to *dest*, which is an absolute address.

WARNING: this function may not work properly if the source MPE is running.

2.5 DMA Functions

2.5.1 _mpedmaregister

`#include <nuon/mutil.h>`
long **_mpedmaregister**(**long** *dmaflags*, **void ***regaddr*, **long** *value*, **int** *mpe*)

Read or write another MPE's register. This is a slightly different interface to **_DMALinear**, which is more convenient for some purposes. *mpe* is the number of the destination MPE (which should *not* be the current MPE). *dmaflags* are the other bus flags to be used by the DMA transfer; normally this should be `0x0001000` for a write and `0x00012000` for a read. *regaddr* is the address of the register to read or write, given as an address in MPE 0's memory map. *value* is the 32 bit value to write into the other MPE's register, and is ignored for read operations.

_mpedmaregister returns either the value it just wrote, or the value read from the other MPE's register.

2.5.2 _raw_plotpixel

```
#include <nuon/dma.h>
```
void **_raw_plotpixel**(**long** *dmaflags*, **void ***baseaddr*, **long** *xinfo*, **long** *yinfo*, **long** *color*)

Plots a rectangle in a single color. This function is basically a direct interface to a direct mode bilinear DMA. *dmaflags* are the flags for the main bus DMA. The read bit must *not* be set, and the flags must be set up for a pixel mode write. **_raw_plotpixel** will itself set the `DIRECT` bit, so it need not be set in *dmaflags*. *baseaddr* is the base address of the area into which rendering is performed; it must be on a 512 byte boundary in SDRAM. *xinfo* and *yinfo* set the x and y coordinates for the rectangle being plotted, as well as the width and height. *xinfo* has the width in its upper 16 bits, and the x position in its lower 16 bits. Similarly, *yinfo* has the height in its upper 16 bits, and the y position in the lower 16 bits. Finally, *color* contains the value to be drawn into the rectangle; this longword will be replicated throughout the area, and will typically be a 32 bit YCrCb color value, 16 bit color plus 16 bit Z, or two 16 bit YCrCb color values (depending on the *dmaflags*). Note that for a 16 bit per pixel output buffer, only the upper 16 bits of *color* will be used. For an 8 bit per pixel output buffer, two pixels at a time (the two specified in the upper 16 bits of *color*) are plotted; unless some sort of dithering is desired, make these two bytes the same in 8 bit per pixel mode.

NOTE: As with all DMA operations, the total amount of data transferred during a single **_raw_plotpixel** operation should be at most 64 long words total. This ensures a predictable latency on the bus, without which some software (for example, MPEG playback) may break, and some BIOS functions will not work.

2.5.3 _synccache

```
#include <nuon/mutil.h>
```
void **_synccache**(*void*)

Synchronizes memory with the MPE's data cache. After this call is made all data is guaranteed to have been stored out to memory. Since the data cache is not a write through cache, it is usually necessary to make either a **_synccache** or **_flushcache** call before doing any DMA operations which must read SDRAM or system bus RAM. **_synccache** differs from **_flushcache** 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 which another MPE has set, you must use **_flushcache** to mark the cache invalid.

2.5.4 _flushcache

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

void **_flushcache**(*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 invoking a function which wishes to make use of local RAM for DMAs or similar purposes. Since the cache is marked as invalid, use of the local RAM for scratch purposes will be safe until a data access occurs which causes a data cache miss.

2.6 Fixed Point Math Functions

2.6.1 DoubleToFix

```
#include <nuon/mutil.h>
```
int **DoubleToFix**(**double** *d*, **int** *shift*)

Converts the floating point number *d* into a fixed point number with *shift* bits of fractional precision. *shift* must be non-negative.

2.6.2 FixToDouble

```
#include <nuon/mutil.h>
```
double **FixToDouble**(**int** *f*, **int** *shift*)

Converts the fixed point number *f*, which has *shift* fractional bits, into a double precision floating point number. *shift* must be non-negative.

2.6.3 FixMul

```
#include <nuon/mutil.h>
```
int **FixMul**(**int** *a*, **int** *b*, **int** *shift*)

Multiplies the fixed point numbers *a* and *b* together, and returns the result shifted right by *shift*. 64 bits are used for the calculation. This "function" is actually implemented as a macro, and is very fast.

2.6.4 FixDiv

```
#include <nuon/mutil.h>
```
int **FixDiv**(**int** *a*, **int** *b*, **int** *shift*)

Divides the fixed point number *a* by the fixed point number *b*. *shift* is the number of fractional bits in *b*. The answer is a fixed point number with the same number of fractional bits as *a*.

2.6.5 FixRecip

```
#include <nuon/mutil.h>
```
long long **FixRecip**(**int** *a*, **int** *fracbits*)

Finds the reciprocal of the fixed point number *a*. *fracbits* is the number of fractional bits in *a*. **FixRecip** returns a 64 bit value: the upper 32 bits is the mantissa of

the reciprocal, and the lower 32 bits is the number of fractional bits in the reciprocal. Note that the input parameter *a* must be a positive number.

2.6.6 FixSinCos

```
#include <nuon/mutil.h>
```
int **FixSinCos**(**int** *angle*, **int ***sinval*, **int ***cosval*)

Calculates both the sine and cosine of an angle. *angle* is a 16.16 fixed point number expressing the angle in rotations (so for example 45 degrees would be `0x2000`). The sine and cosine of *angle* are computed, and their values as 2.30 fixed point numbers are placed in the locations pointed to by *sinval* and *cosval* respectively. The sine is also returned as the result of **FixSinCos**.

2.6.7 FixSqrt

```
#include <nuon/mutil.h>
```
int **FixSqrt**(**int** *x*, **int** *fracbits*)

Calculates the fixed point square root of the fixed point number *x*. *fracbits* is the number of fractional bits in *x*. The answer is returned with the same number of fractional bits.

2.6.8 FixRSqrt

```
#include <nuon/mutil.h>
```
int **FixRSqrt**(**int** *x*, **int** *xbits*, **int** *rbits*)

Calculates the reciprocal of the fixed point square root of the fixed point number *x*. *xbits* is the number of fractional bits in *x*. The answer is returned with *rbits* fractional bits.

2.7 SDRAM memory functions

2.7.1 SDRAMAlloc

```
#include <nuon/sdram.h>
```
void * **SDRAMAlloc**(**unsigned long** *size*)

Allocates *size* bytes of memory in SDRAM. This function is a direct interface to the BIOS **_MemAlloc** function, and shares the same bugs; many older BIOS versions do not mark memory used by the program's COFF file, so **SDRAMAlloc** may return this memory.

2.7.2 SDRAMFree

```
#include <nuon/sdram.h>
```
void **SDRAMFree**(**void ***ptr*)

Frees memory in SDRAM which was previously allocated by **SDRAMAlloc** (or **_MemAlloc**).

2.7.3 SDRAMInit

```
#include <nuon/sdram.h>
```
void **SDRAMInit**(**void ***_startaddr_, **unsigned long** _size_)

This is an obsolete function which no longer has any effect. It is now simply a do-nothing stub provided for compatibility with some old source code.

2.8 Miscellaneous Functions

2.8.1 _GetLocalVar

```
#include <nuon/mutil.h>
```
int **_GetLocalVar**(**int** *&addr*)

This is a macro for fetching the contents of a variable (or register) in an MPE's local memory. Since it is a macro, the name of the variable (rather than a pointer to it) may be passed as the argument. Use of this macro is recommended, because it works around a bug in the cache which can cause the MPE to hang if a cached memory access is followed immediately by an uncached access (*i.e.* to local memory or register). Note that variables in the `intdata` section are placed in local memory by default, but variables in all other sections default to system ram, which is cached memory.

2.8.2 _SetLocalVar

```
#include <nuon/mutil.h>
```
int **_SetLocalVar**(**int** *&addr*, **int** *val*)

This is a macro for setting the contents of a variable (or register) in an MPE's local memory. Since it is a macro, the name of the variable (rather than a pointer to it) may be passed as the argument. Use of this macro is recommended, because it works around a bug in the cache which can cause the MPE to hang if a cached memory access is followed immediately by an uncached access (*i.e.* to local memory or register). Note that variables in the `intdata` section are placed in local memory by default, but variables in all other sections default to system ram, which is cached memory.

2.8.3 msprintf

```
#include <nuon/msprintf.h>
```
int **msprintf**(**char ***_buf_, **const char ***_fmt_, ...)

A simple version of **sprintf** which may be used for formatting when the full power of the standard C library is not needed. **msprintf** only supports integer output formats. It will output at most *SPRINTF_MAX* characters into the given buffer, and returns the number of characters actually output.

Using **msprintf** instead of **sprintf** may reduce the size of your executable if you use no other facilities from the standard I/O library.

2.8.4 DebugWS

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

void **DebugWS**(**long** *dmaflags*, **void ***baseaddr*, **int** *xpos*, **int** *ypos*, **long** *color*, **const char ***string*)

Write a message into a frame buffer. *dmaflags* and *baseaddr* are the DMA flags used for writing into the buffer and the base address of the frame buffer, respectively. *xpos* and *ypos* are the x and y coordinates for the upper left hand corner of the string. *color* is the color used to draw the string; the interpretation of this depends on the pixel type in *dmaflags*. *string* is the (zero terminated) ASCII string to write.

Note that the font used for **DebugWS** is quite ugly, and does not contain all punctuation characters.

Index

_CommSend, 1
_DMALinear, 7
_GetLocalVar, 11
_MemAlloc, 10
_SetLocalVar, 11
_TimeElapsed, 5
_comm_recv, 4
_comm_recv_query, 4
_comm_send, 4
_flushcache, 8, 9
_mpedmaregister, 7, 8
_raw_plotpixel, 8
_synccache, 8

CommRecv, 2, 3
CommRecvInfo, 3, 4
CommRecvQuery, 2, 3
CommRecvQueryInfo, 3, 4
CommSend, 1, 2
CommSendInfo, 2–4
CommSendRecv, 2–4
communication bus, 2, 3
CopyFromMPE, 7
CopyToMPE, 7

DebugWS, 12
DoubleToFix, 9

FixDiv, 9
FixMul, 9
FixRecip, 9
FixRSqrt, 10
FixSinCos, 10
FixSqrt, 10
FixToDouble, 9

GetTimer, 5

msprintf, 11

ReadMPERegister, 5

SDRAMAlloc, 10
SDRAMFree, 10
SDRAMInit, 11
sprintf, 11
StartMPE, 6, 7
StopMPE, 6

video, 5
VidSetup, 5

WaitMPE, 6
WriteMPERegister, 6