

GRAPHICS CONTROLLERS

MB88F332

'INDIGO' BYTE ORDER

APPLICATION NOTE

Graphics Competence Center
'Indigo' Software Starter Kit



Revision History

Date	Issue
13-Oct-2008	Revision 0.10, Th. Betker: First draft.
03-Dec-2008	Revision 0.20, Th. Betker: Updated "GdcCom API" section. Added text for "Run-Length Compression" section. Extended "System Setup" chapter.
06-Apr-2009	Revision 0.21, Th. Betker: Include ES3 in "Default Setup" section.

This document contains 16 pages.

Warranty and Disclaimer

To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH restricts its warranties and its liability for **all products delivered free of charge** (eg. software include or header files, application examples, target boards, evaluation boards, engineering samples of IC's etc.), its performance and any consequential damages, on the use of the Product in accordance with (i) the terms of the License Agreement and the Sale and Purchase Agreement under which agreements the Product has been delivered, (ii) the technical descriptions and (iii) all accompanying written materials. In addition, to the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH disclaims all warranties and liabilities for the performance of the Product and any consequential damages in cases of unauthorised decompiling and/or reverse engineering and/or disassembling. **Note, all these products are intended and must only be used in an evaluation laboratory environment.**

1. Fujitsu Microelectronics Europe GmbH warrants that the Product will perform substantially in accordance with the accompanying written materials for a period of 90 days from the date of receipt by the customer. Concerning the hardware components of the Product, Fujitsu Microelectronics Europe GmbH warrants that the Product will be free from defects in material and workmanship under use and service as specified in the accompanying written materials for a duration of 1 year from the date of receipt by the customer.
2. Should a Product turn out to be defect, Fujitsu Microelectronics Europe GmbH's entire liability and the customer's exclusive remedy shall be, at Fujitsu Microelectronics Europe GmbH's sole discretion, either return of the purchase price and the license fee, or replacement of the Product or parts thereof, if the Product is returned to Fujitsu Microelectronics Europe GmbH in original packing and without further defects resulting from the customer's use or the transport. However, this warranty is excluded if the defect has resulted from an accident not attributable to Fujitsu Microelectronics Europe GmbH, or abuse or misapplication attributable to the customer or any other third party not relating to Fujitsu Microelectronics Europe GmbH.
3. To the maximum extent permitted by applicable law Fujitsu Microelectronics Europe GmbH disclaims all other warranties, whether expressed or implied, in particular, but not limited to, warranties of merchantability and fitness for a particular purpose for which the Product is not designated.
4. To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH's and its suppliers' liability is restricted to intention and gross negligence.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES

To the maximum extent permitted by applicable law, in no event shall Fujitsu Microelectronics Europe GmbH and its suppliers be liable for any damages whatsoever (including but without limitation, consequential and/or indirect damages for personal injury, assets of substantial value, loss of profits, interruption of business operation, loss of information, or any other monetary or pecuniary loss) arising from the use of the Product.

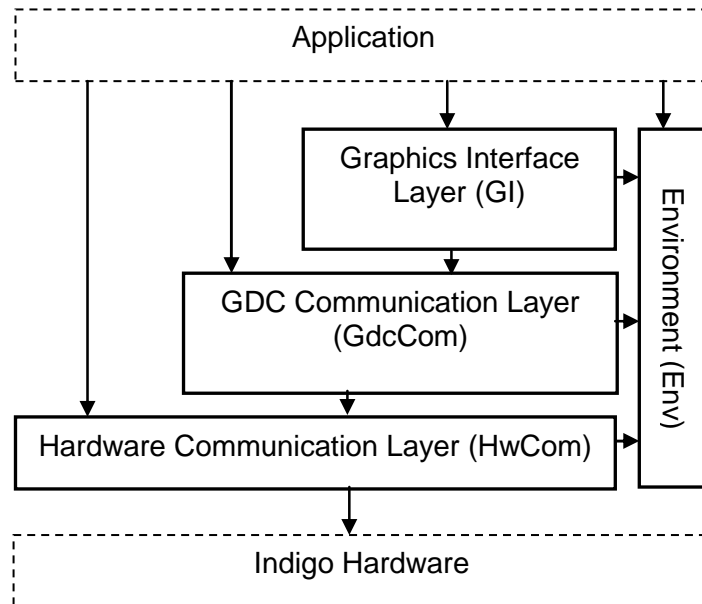
Should one of the above stipulations be or become invalid and/or unenforceable, the remaining stipulations shall stay in full effect

Contents

REVISION HISTORY	2
WARRANTY AND DISCLAIMER	3
CONTENTS	4
1 INTRODUCTION	5
2 ENDIANNESS	6
2.1 'Indigo' Hardware Modules	6
2.2 Little-Endian and Big-Endian	6
3 SYSTEM SETUP	7
3.1 Default Setup	7
3.2 Recommended Setup	7
4 HWCOP API	9
4.1 Transfer Protocols	9
4.2 Full-Register Access	9
4.3 Sub-Register Access	10
5 OTHER APIS	12
5.1 GdcCom API	12
5.2 ARH Driver API	12
6 INDIRECT REGISTER ACCESS	13
6.1 RAM and Flash Memory	13
6.2 Command Sequencer	13
6.3 Configuration FIFO	13
7 PATTERN DATA	14
7.1 Sprite Engine	14
7.2 Run-Length Compression	15
8 APPENDIX	16
8.1 Abbreviations	16
8.2 References	16

1 Introduction

This application note discusses the endianness of the MB88F332 'Indigo' registers, and how it is handled in the 'Indigo' Software Starter Kit.



Basically, the HwCom API and GdcCom API have been designed to read and write registers in “host byte order”, as `uint32`, `uint16`, or `uint8`. This allows for simple, clear and portable register operations, independent of the endianness of the 'Indigo' hardware modules, or the byte order of the host running the application.

However, issues arise when accessing parts of a register, e.g., when trying to read a word register by halfwords, or by bytes: The addressing differs, depending on the endianness of the hardware module. Also, indirect register access by Configuration FIFO or Command Sequencer has to be considered.

In addition, we cover related topics such as byte and bit order in Sprite Engine pattern data, or in input data for Run-Length Decompression.

2 Endianness

As specified in the MB88F332 hardware manual [1], most 'Indigo' hardware modules such as the Sprite Engine are little-endian with 32-bit registers. However, there are also some legacy hardware modules which are big-endian, with 8-bit, 16-bit, and a few 32-bit registers.

2.1 'Indigo' Hardware Modules

The following legacy hardware modules in 'Indigo' are big-endian:

- Clock Modulator
- Stepper Motor Controller
- Sound Generator
- I²C Controller
- USART (LIN/FIFO)
- Programmable Pulse Generators
- A/D Converter
- Reload Timer

All other modules are little-endian. In particular, all AHB bus masters are little-endian: Host Interface, Remote Handler, Command Sequencer, and Configuration FIFO.

Note: The External Interrupt unit is not a legacy module, although the MB88F332 specification may indicate otherwise. The module is little-endian, and all registers are 32-bit (with bits 31..24 being reserved).

2.2 Little-Endian and Big-Endian

Endianness is meant here from the perspective of the AMBA AHB [2]. Little-endian and big-endian AHB slaves or masters simply use different byte lanes on the bus for halfword and byte transfers.

Implicitly, endianness also determines how parts of a slave's register are addressed (e.g., halfword access to a word register). The following two tables show the general behavior for 32-bit and 16-bit registers:

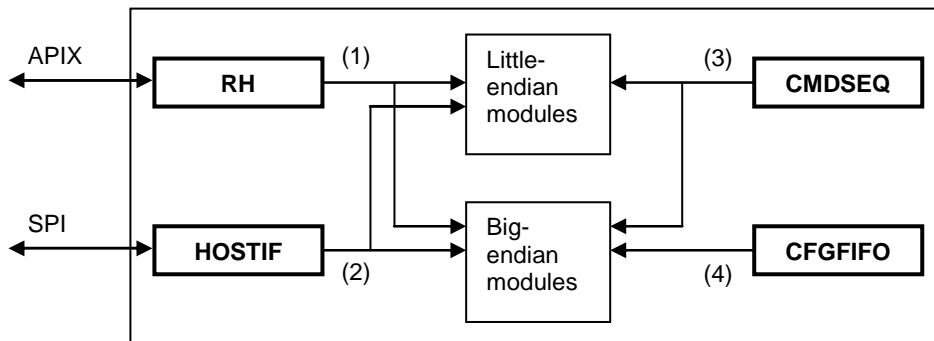
(32-bit register)	little-endian	big-endian
value32	word(addr)	word(addr)
(value32>>16)&0xffff	halfword(addr+2)	halfword(addr+0)
(value32>> 0)&0xffff	halfword(addr+0)	halfword(addr+2)
(value32>>24)&0xff	byte(addr+3)	byte(addr+0)
(value32>>16)&0xff	byte(addr+2)	byte(addr+1)
(value32>> 8)&0xff	byte(addr+1)	byte(addr+2)
(value32>> 0)&0xff	byte(addr+0)	byte(addr+3)
(16-bit register)	little-endian	big-endian
value16	halfword(addr)	halfword(addr)
(value16>> 8)&0xff	byte(addr+1)	byte(addr+0)
(value16>> 0)&0xff	byte(addr+0)	byte(addr+1)

Note that we avoid talking about "byte order" here, as this perspective does not really help. Instead, we use a logical view of words or halfwords which does not depend on a specific representation in host memory. E.g., (value32>>24)&0xff (shift and mask) is always the most significant byte of a value32 word, for any internal representation on the host.

3 System Setup

In order to handle endianness issues, the 'Indigo' hardware contains a set of data swappers (which are documented in the specification, and configured as part of Chip Control), and a few address inverters (which are not documented).

The following diagram shows the access paths that have to be considered when discussing setups for data swapping and address inversion:



3.1 Default Setup

In the current chip versions (up to ES3), the initial setup of data swappers and address inverters at 'Indigo' power-up does not match all requirements for operation. Access to halfword and byte registers by Host Interface and Command Sequencer does not work as expected, not does sub-register access to little-endian modules by the Remote Handler.

When the default setup is used, the software has to compensate this behaviour by implementing its own address inversion for some access paths:

access path	little-endian module	big-endian module
(1) Remote Handler [APIX]	Invert	Normal
(2) Host Interface [SPI]	Normal	Invert
(3) Command Sequencer	Normal	Invert
(4) Configuration FIFO	--	Normal

Address inversion means that for 8-bit access, the last two bits of the address are toggled (00↔11, 01↔10), and for 16-bit access, the next to last bit (00↔10); for 32-bit access, no address bits are toggled.

In particular, address inversion must always be applied when addressing 8-bit and 16-bit registers in a command list (OSETREG), regardless of the communication link used by the host (APIX or SPI).

3.2 Recommended Setup

When the address inverters and data swappers are set up properly, no address inversion in software is needed (for any access path).

The following registers have to be set, in this order, for the recommended setup:

Register	Address	Value	
CCNT.ClockEnable	0x00010814	0xFFFFFFFF	// enable all clocks
CCNT.DataSwapCtr1	0x00010820	0x00000000	// set up swappers
CCNT.DataSwapCtr2	0x00010824	0x00000000	
CCNT.DataSwapCtr3	0x00010828	0x00000000	
CCNT.DataSwapCtr4	0x0001082C	0x00000505	
CCNT.0xE00	0x00010E00	0x5E5F5E76	// set up inverters
CCNT.0xE30	0x00010E30	0x00000000	
CCNT.0xE00	0x00010E00	0x00000000	

If you are using HwCom, GdcCom, or GI, this is done by `HwComSystemSetup()`. This function is called by `GdcComOpen()`, which in turn is called by `GiOpen()`.

If you do not use HwCom, you should set the registers above as part of your reset command sequence stored in flash memory; the SETREG commands would be as follows:

```
02010000 00010814 FFFFFFFF
02040000 00010820 00000000 00000000 00000000 00000505
02010000 00010E00 5E5F5E76
02010000 00010E30 00000000
02010000 00010E00 00000000
```

Your installation of Fujitsu GDC Studio should come with an `indigo-system-setup.gdcseq` register sequence which can be used as a starting point for generating this command sequence.

4 HwCom API

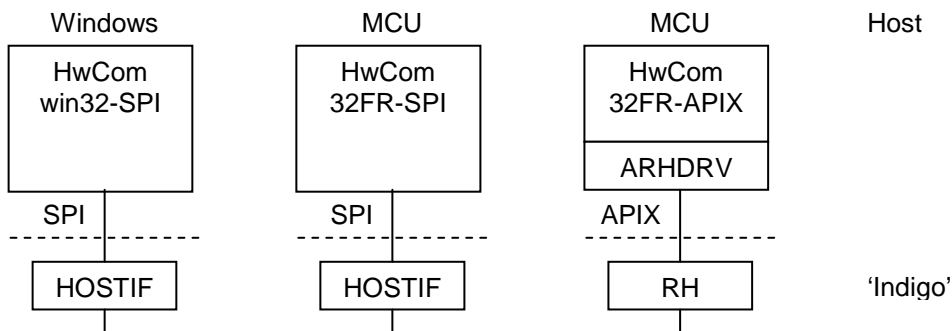
This chapter presents the low-level view of direct register access via Host Interface and Remote Handler. For indirect operations, please see the chapter “Indirect Register Access”.

We are assuming here that the system has been properly set up by `HwComSystemSetup()` [immediately after `HwComOpen()`] before calling any `HwComWriteXxx()` or `HwComReadXxx()` functions.

4.1 Transfer Protocols

Currently, HwCom supports three different configurations; everything discussed in the next sections applies to all three of them:

- win32-SPI Windows host, using SPI (by Aardvark) and Host Interface
- 32FR-SPI MCU host, using SPI and Host Interface
- 32FR-APIX MCU host, using APIX and Remote Handler



Note that Windows is little-endian, while the MCU is big-endian; this is the host byte order, which must not be confused with the endianness of the ‘Indigo’ registers.

However, the definition of the protocols used to transfer addresses and data to the Host Interface and Remote Handler modules is independent of the host byte order. Please see the MB88F332 specification for details of the HOSTIF and RH protocols.

HwCom handles reading and writing the protocol messages, and converting addresses and data between host byte order and transfer byte order. For the HOSTIF protocol, this is done explicitly in the HwCom implementation.

For the RH protocol, HwCom uses the ARH Driver from Fujitsu Microelectronics. The `HwComReadXxx()` and `HwComWriteXxx()` functions map directly to `ArhIf_SyncReadScalar()` and `ArhIf_SyncWriteScalar()`; please see the section “ARH Driver API” in the next chapter.

4.2 Full-Register Access

The register values returned by the `HwComReadXxx()` functions are in host byte order, and the register values passed to the `HwComWriteXxx()` functions are also in host byte order. In other words, you can use the appropriate shift and mask operations to get and set bits in the register value, in conformance with the MB88F332 specification. This works both for little-endian and big-endian registers, and for any host byte order.

The following example shows how the SPEBG register (Background Color Register) of the Sprite Engine is set to the “indigo” color (RVAL = 0x2e, GVAL = 0x08, BVAL = 0x54):

```
enum INDIGO_XFER_STATUS_ENUM_T xferStatus;

if (!HWComWriteRegister32(0, &xferStatus, 0x20000, 0x2e0854)) {
    /* handle error */
}
```

The following example shows how the SPETP0 register (Transparent Color Register 0, for indirect color mode) is read from the Sprite Engine, and its fields are extracted (TP1B, TP2B, TP4B, TP8B):

```
uint32 spetp0, tp1b, tp2b, tp4b, tp8b;
enum INDIGO_XFER_STATUS_ENUM_T xferStatus;

if (HWComReadRegister32(0, &xferStatus, 0x20004, &spetp0)) {
    tp1b = (spetp0>>14)&0x1;
    tp2b = (spetp0>>12)&0x3;
    tp4b = (spetp0>>8)&0xf;
    tp8b = (spetp0>>0)&0xff;
}
```

As demonstrated above, you should always use shift and mask operations to access parts of a word. Do not try to handle an `uint32` value as `uint16[2]` or `uint8[4]`, as this is definitely not portable.

4.3 Sub-Register Access

Accessing a word register by halfwords or bytes, or accessing a halfword register by bytes, works just like full-register access, except that addressing depends on the endianness of the 'Indigo' module (the tables below are similar to the tables in the "Endianness" chapter):

(32-bit register)	little-endian	big-endian
31:0	word(addr)	word(addr)
31:16	halfword(addr+2)	halfword(addr+0)
15:0	halfword(addr+0)	halfword(addr+2)
31:24	byte(addr+3)	byte(addr+0)
23:16	byte(addr+2)	byte(addr+1)
15:8	byte(addr+1)	byte(addr+2)
7:0	byte(addr+0)	byte(addr+3)
(16-bit register)	little-endian	big-endian
15:0	halfword(addr)	halfword(addr)
15:8	byte(addr+1)	byte(addr+0)
7:0	byte(addr+0)	byte(addr+1)

For big-endian modules, the access methods and addresses are usually listed in the specification. The following example shows how to set the SGAR register (Amplitude Data register) of the Sound generator module by an 8-bit write, and by a 16-bit write (bits 7:0 of the register are reserved):

```
uint8 sgar = 240;

(void) HWComWriteRegister8(0, &xferStatus, 0x19a, sgar);
(void) HWComWriteRegister16(0, &xferStatus, 0x19a, sgar<<8);
```

For little-endian modules, the access is usually not listed. In general, you may assume that halfword and byte access is supported unless stated otherwise. The following example shows how to read the PDR0 register (Port Data Register 0) of the GPIO module by a 32-bit read, and by an equivalent 8-bit read (bits 31:8 are reserved):

```
uint32 pdr0_reg;  
uint8 pdr0;  
  
(void) HwComReadRegister32(0, &xferStatus, 0x15000, &pdr0_reg);  
pdr0 = (uint8) (pdr0_reg&0xff);  
  
(void) HwComReadRegister8(0, &xferStatus, 0x15000, &pdr0);
```

Note that some little-endian modules may restrict halfword and byte access to their 32-bit registers; e.g., the Sprite Engine supports it only for the Sprite Attribute Table registers. Overall, full-register (32-bit) access is recommended.

5 Other APIs

In general, GdcCom is used as a high-layer interface for HwCom. However, there is also the 32FR-APIX case where no HwCom is used, but the underlying ARH Driver interface.

5.1 GdcCom API

GdcCom uses HwCom, so GdcCom full-register access works as described for HwCom. Note that `GdcComOpen()` calls `HwComOpen()` and `HwComSystemSetup()`, so you don't have to take care of this.

GdcCom does not have an API for sub-register access. The register and field macros in `*_hal.h` assume that you always use 32-bit access for 32-bit registers, 16-bit access for 16-bit registers, and 8-bit access for 8-bit registers.

For the little-endian modules, there are only 32-bit registers. For the big-endian modules, it is not always obvious which access is intended by GdcCom's symbolic constants. You should check the fourth parameter of the register and field macros, *strideNextElement*, in this case; since there are no arrayed registers for big-endian modules, this is the register size, in bytes (4 for 32-bit, 2 for 16-bit, 1 for 8-bit). The following table can be used as a guideline:

module	registers	access
UART	all registers	16-bit
SMC	PWC and PWS registers SPWS_SPWC registers	16-bit 32-bit
I2C	all registers	16-bit
PPG	all registers	16-bit
SOUND	all registers	16-bit
ADC	all registers	16-bit
RLT	all registers	16-bit
CLOMO	all registers	8-bit

Note: The SMC shadow registers (SPWC and SPWS registers) are not documented.

5.2 ARH Driver API

In the 32FR-APIX configuration, where the application is running on the MCU and register access is via the RH protocol, some customers may want to use the ARH Driver directly instead of the HwCom API wrapper.

In this case, `ArhIfSyncReadScalar()` [or `ArhIfReadScalar()`] is used instead of `HwComReadXxx()`, and `ArhIfSyncWriteScalar()` [or `ArhIfWritecalar()`] is used instead of `HwComWriteXxx()`. Register (and sub-register) access works as for HwCom then, provided that the system has been set up properly.

The system can be set up either by a reset command sequence stored in flash, or you have to implement a replacement for `HwComSystemSetup()` in your application.

6 Indirect Register Access

This chapter deals with indirect register access by Command Sequencer and Configuration FIFO, as opposed to direct access via Host Interface and Remote Handler.

The idea is that setting up a command sequence or programming a FIFO follows the general HwCom principles, and that addresses and data are handled the same way as for direct access. Again, the assumption here is that the system has been set up properly; please see the chapter "System Setup".

In this context, we also discuss memory access, as a complement to register access.

6.1 RAM and Flash Memory

The controller for 'Indigo's' built-in SRAM is little-endian with 32-bit access, and writing or reading data to it is straightforward. In general, 32-bit writes are used for pattern data or command sequences (e.g., `HwComWriteRegister32()`, `HwComWriteBlock32()`), but 16-bit or 8-bit access works as well.

The Embedded Flash Memory module is also little-endian, with 32-bit registers, but flash programming is by halfwords. In order to write a 32-bit word `value32` to flash address `addr`, write `(value32>>0)&0xffff` to `addr`, and `(value32>>16)&0xffff` to `addr+2` (using the proper programming sequences).

Word access can be used to read the flash, though; reading halfwords or bytes also works.

6.2 Command Sequencer

A command sequence consists of 32-bit words in RAM or flash memory which is processed by the Command Sequencer module. The format of the commands is described in the MB88F332 specification.

The command words are written to 'Indigo' RAM by 32-bit writes (e.g., `HwComWriteRegister32()` or `HwComWriteBlock32()`), and are provided in host byte order. The following example writes a simple command list to 'Indigo's' 8K RAM block:

```
uint32 cmdList[5] = {
    0x01010003,                /* WAIT VSync */
    0x02010000, 0x20000, 0x2e0854, /* SETREG SPEBG=indigo */
    0xffffffff                /* END */
};

(void) HwComWriteBlock32(0, &xferStatus, 0x50000, cmdList, 5);
```

16-bit and 8-bit register access works as well, using `OSETREG` instead of `SETREG`; the following example is for a 16-bit register write:

```
uint32 cmdList[4] = {
    0x03010000, 0x19a, 0x0000F000 /* OSETREG SGAR=240 */
    0xffffffff                /* END */
};

(void) HwComWriteBlock32(0, &xferStatus, 0x50000, cmdList, 4);
```

The command lists above can also be programmed into flash memory, using GDC Studio or your own flash routines.

6.3 Configuration FIFO

(to be supplied)

7 Pattern Data

This chapter discusses the pattern data used by the Sprite Engine. The host can store the pattern data uncompressed (and write it directly to 'Indigo' RAM or flash), or compressed (and send it to Run-Length Decoding, which writes the uncompressed data to 'Indigo' RAM).

7.1 Sprite Engine

Uncompressed pattern data is organized in 32-bit words, as this is what the Sprite Engine reads from 'Indigo' RAM or flash memory. If you use the Image Manager of Fujitsu GDC Studio, choose "Indigo 32 Bit" as "Organization" when generating source files from an image.

The GI layer uses 32-bit block writes to transfer the data to 'Indigo' RAM since this is the most effective way. Remember that the `uint32[]` data is in host byte order, and that you should not try to interpret it as `uint16[]` or `uint8[]`.

The "Sprite Engine" chapter in the MB88F332 specification explains how pixels are packed into the pattern data. Translated to C pseudo-code, the pixel at position (x, y) can be accessed as follows (sw and sh are sprite width and sprite height):

```
extern uint32 data[], sw, sh;
extern enum INDIGO_COLOR_FORMAT_ENUM_T format;

uint32 getPixel(uint32 x, uint32 y)
{
    uint32 p = x + sw*y;

    switch (format) {
        case INDIGO_COLOR_FORMAT_1BPP:
            return ((data[p/32] >> ((p%32)*1)) & 0x1);
        case INDIGO_COLOR_FORMAT_2BPP:
            return ((data[p/16] >> ((p%16)*2)) & 0x3);
        case INDIGO_COLOR_FORMAT_4BPP:
            return ((data[p/8] >> ((p%8)*4)) & 0xf);
        case INDIGO_COLOR_FORMAT_8BPP:
            return ((data[p/4] >> ((p%4)*8)) & 0xff);
        case INDIGO_COLOR_FORMAT_RGB565:
        case INDIGO_COLOR_FORMAT_ARGB1555:
            return ((data[p/2] >> ((p%2)*16)) & 0xffff);
        case INDIGO_COLOR_FORMAT_ARGB8888:
            return data[p];
        default:
            return 0x00000000;
    }
}
```

For indirect color formats, the `pixel` value is a color index. For indirect color formats, you can extract the ARGB values by the appropriate shift and mask operations on `pixel`; e.g, for ARGB1555:

```
uint32 pixel; /* format = INDIGO_COLOR_FORMAT_ARGB1555 */
uint32 alpha = (pixel>>15)&0x1;
uint32 red    = (pixel>>10)&0x1f;
uint32 green  = (pixel>> 5)&0x1f;
uint32 blue   = (pixel>> 0)&0x1f;
```

Note that all the above applies to alpha tables as well, except that only 4bpp and 4bpp indirect color formats are supported for alpha tables.

7.2 Run-Length Compression

RLD input and output data are bit streams. The output stream consists of pixels (1, 2, 4, 8, 16, 24, or 32 bits), aligned to pixel boundaries.

The input stream consists of command bytes (8 bits) and pixels. Alignment is only to bits, not to bytes, pixels, or words, and there is no padding. At 2 bits per pixel, a command byte may start at a 2-bit boundary. At 32 bits per pixel, a pixel may start at an 8-bit boundary.

Both input and output data are organized in 32-bit words. Within a word, the most significant bit comes first, and the least significant bit comes last. The following code shows how to read a single bit from a stream:

```
uint32 readBit( uint32 *stream, uint32 pos)
{
    return ((stream[pos/32]>>(31-pos%32))&1);
}
```

The following code reads an n-bit integer, n = 0..32 (this code just serves to illustrate the idea, and is not optimized):

```
uint32 readInteger( uint32 *stream, uint32 pos, uint32 n)
{
    value = 0;
    for (j = 0; j < n-1; j++)
        value = (value<<1) | readBit(words, pos+j);
    return value;
}
```

Note that the RLD pixel order in its input or output words (most to least significant bits) is not the same as SPE's pixel order (least to most significant bits). So you have to take care when compressing some pixel data to create RLD input: The pixels need to be processed in RLD order (as above), not in SPE order.

In practice, this means that the pixels should fit exactly into 32 bit words, i.e., the number of pixel bits must be a multiple of 32. Otherwise, the pixels in the last output word are not at the positions expected by the Sprite Engine.

This restriction seems to be a good idea anyway. Otherwise, the data in the last word written by the RLD module may be corrupted (apparently, the data is not shifted to the most significant bits, but stays in the least significant bits). The condition is mostly relevant for 24 bpp pixel data (the number of words must be a multiple of 3). For 1, 2, 4, 8, 16, 32 bpp, it just means that the pixels make up an integral number of words.

8 Appendix

8.1 Abbreviations

- AHB Advanced High-Performance Bus; part of the AMBA specification.
AMBA Advanced Microcontroller Bus Architecture, by ARM Limited.
API Application Programming Interface.

8.2 References

- [1] *MB88F332, LSI Product Specification*. Edition 0.97, August 2008. Fujitsu Limited.
[2] *AMBA™ Specification*. Rev. 2.0, 13th May 1999. ARM Limited.