Arm® Mali™-IV009 C Model Integration Guide

Version r0p0-00rel0 / Revision r0p0

Revision Information

Date	Issue	Confidentiality	Change
16/02/2018	EAC	Confidential	Initial release
14/09/2018	REL	Confidential	First release

Confidential Proprietary Notice

This document is CONFIDENTIAL and any use by you is subject to the terms of the agreement between you and Arm or the terms of the agreement between you and the party authorised by Arm to disclose this document to you.

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information: (i) for the purposes of determining whether implementations infringe any third party patents; (ii) for developing technology or products which avoid any of Arm's intellectual property; or (iii) as a reference for modifying existing patents or patent applications or creating any continuation, continuation in part, or extension of existing patents or patent applications; or (iv) for generating data for publication or disclosure to third parties, which compares the performance or functionality of the Arm technology described in this document with any other products created by you or a third party, without obtaining Arm's prior written consent.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20348

Confidentiality Status

This document is Confidential. This document may only be used and distributed in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

The information in this document is for a product under development and is not final.

Contents

Preface

This preface introduces the Arm® C Model Integration Guide. It contains the following sections:

- About this book.
- Feedback.

About this book

This book is for the Arm Mali-IV009 C Model Integration.

Implementation obligations

This book is designed to help you implement an Arm product. The extent to which the deliverables may be modified or disclosed is governed by the contract between ARM and Licensee. There may be validation requirements, which if applicable will be detailed in the contract between Arm and Licensee and which if present must be complied with prior to the distribution of any silicon devices incorporating the technology described in this document. Reproduction of this document is only permitted in accordance with the licences granted to Licensee.

Arm assumes no liability for your overall system design and performance, the verification procedures defined by Arm are only intended to verify the correct implementation of the technology licensed by Arm, and are not intended to test the functionality or performance of the overall system. You or the Licensee will be responsible for performing any system level tests.

You are responsible for any applications which are used in conjunction with the Arm technology described in this document, and to minimize risks adequate design and operating safeguards should be provided for by you. Arm's publication of any information in this document of information regarding any third party’s products or services is not an express or implied approval or endorsement of the use thereof.

Product revision status

The rmpn identifier indicates the revision status of the product described in this book, for example, r1p2, where:

rm Identifies the major revision of the product, for example, r1.

pn Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This book is for system designers, system integrators, and verification engineers who are designing a System-on-Chip (SoC) device that uses the Mali-IV009 Image Signal Processor.

Using this book

This book is organized into the following sections:

Section	Description
Introduction	Introduces the C Model integration.
Delivery package	Lists the files contained in the delivery package.
Library API	Describes the various API functions available.
Build process	Provides brief information about the build process.
Executable command line parameters	List the executable command line parameters.
Script file format	Provides information about the structure and syntax of the Script file.
Examples	Provides some code examples.
Frame file format	Provides information about Frame files and how to create a Frame file.

Typographic conventions

Font/text type	What it means
italic	Introduces special terminology, denotes cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`

Terms and abbreviations

Term/Abbreviation	What it means
API	Application Program Interface
AXI	Advanced eXtensible Interface
CPU	Central Processor Unit
CRC	Cyclic Redundancy Check
DDR	Double Data Rate
DMA	Direct Memory Access
DS	Down Scaler
FR	Full Resolution
HDL	Hardware Description Language
HW	Hardware
IP	Intellectual Property
ISP	Image Signal Processor
LUT	Look Up Table
OS	Operating System
RTL	Register Transfer Language
SW	Software

Additional reading

Information published by Arm and by third parties.

See http://infocenter.arm.com for access to Arm documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to errata@arm.com. Give:

- The title.
- The number ARM-EPM-137521.
- The page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note: Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

1 Introduction

Intellectual Property (IP) core models are designed for Register Transfer Language (RTL) verification or evaluation within a full system. They can be used directly through an Application Program Interface (API) or by parsing test scripts. Test script files are designed for automatic simulation tests for both Hardware Description Language (HDL) and C++ models. They specify regression tests for an independent module or combination of modules like the Image Signal Processor (ISP) top level.

2 Delivery package

The C model delivery package contains the following files in the folder `mali_iv009/logical/models/c_model/`:

File name	Description
`arm_model_api.h`	Library API (see section Library API for details).
`libarm_model_Lloyd_R0P0.so`	Dynamic library for Linux.
`Lloyd_R0P0`	Executable for Linux.

3 Library API

All API functions are described in the file `arm_model_api.h` with comments and explanations.

3.1 Error values

This section describes the error values returned by the error API function:

```
typedef enum {
    ARM_MODEL_SUCCESS,                // success - no errors
    ARM_MODEL_ERROR_NO_MEMORY,           // not enough memory in the system
    ARM_MODEL_ERROR_ADDRESS_NOT_ALIGNED, // address must be aligned at 32-bits boundary
    ARM_MODEL_ERROR_WRONG_INSTANCE,      // required instance does not exist in IP
    ARM_MODEL_ERROR_CONFIG,              // configuration read/write error
    ARM_MODEL_ERROR_UNKNOWN              // unknown error, read stderr for more information
} arm_model_error_t;
```

3.2 Structure

Structure defines image frames for processing and modules output dump.

```
typedef struct {
    uint16_t width;       // image width in pixels
    uint16_t height;      // image height in pixels
    uint16_t planes;      // number of planes (for example: 1 - RAW, 3 - RGB)
    uint16_t depth;       // actual number of bits per pixel per plane
    size_t data_size;     // size of data array in bytes
    data_t* data;         // array of pixels in layout: plane x X x Y, where X - width coordinate,
Y - height coordinate
    // for example 3x2 RGB:
    // [R00 R01 R02] [G00 G01 G02] [B00 B01 B02]
    // [R10 R11 R12] [G10 G11 G12] [B10 B11 B12]
    // would be: [R00 G00 B00 R01 G01 B01 R02 G02 B02 R10 G10 B10 R11 G11 B11 R12 G12 B12]

} frame_t;

typedef struct {
    char module[MAX_MODULE_NAME];  // name of the module which output that is captured
    frame_t frame;                 // dump output in this frame. Data must be prepared in advance
} dump_t;
```

3.3 Build information

This function returns a pointer to the build information string.

```
const char * arm_model_build_info();
```

Example of the returned build information:

```
Build info:
Git id: 0397dd6591a006ad8f464e319a5d03badf2c9890
Host OS: Linux-4.4.0-101-generic x86_64
Compiler: GNU 5.4.1 /usr/bin/c++
Build type: Debug
```

3.4 External memory access

Callback functions are provided for external memory access.

When set, these callback functions provide access to memory for AXI modules in the IP core. Usually they read from and write to a specific memory location imitating AXI bursts.

- **buffer** - pointer to memory with read/write data.
- **address** - start address of external memory (usually AXI word aligned).
- **size** - data to be transferred in bytes (usually multiple of AXI word).

```
typedef void (*memory_read_cb)(uint8_t* buffer, uint32_t address, uint32_t size);
typedef void (*memory_write_cb)(const uint8_t* buffer, uint32_t address, uint32_t
size);
```

3.5 Set memory callback functions

The callback functions provided are used for access to external memory by modules which use the AXI bus. If the callbacks are set to NULL, the IP core uses an internal memory representation.

- **read_cb** - function for reading from external memory (might be NULL).
- **write_cb** - function for writing to external memory (might be NULL).

```
arm_model_error_t arm_model_set_memory_callbacks(memory_read_cb read_cb,
memory_write_cb write_cb);
```

3.6 Process script file

This function processes the images and configuration specified in a script file. See section Script file format for a description of the script format. The function arguments and code are listed as follows:

- **infile** - name of the script file
- **outfile** - name of the file where read values are stored
- **debug_dump** - if non-zero dump output data from all modules
- **trace_dump** - show data processing path in stdout.

```
arm_model_error_t arm_model_process_script(const char* infile, const char*
outfile, int debug_dump, int
trace_dump);
```

3.7 Read data from configuration space

All transactions are 32-bits wide. A description of the related product registers and memory address map are provided with each product documentation set in the form of a register map.

- **data** - pointer to buffer to save data.
- **address** - start address aligned to 32-bit boundary.
- **items** - number of 32-bit words to read.

```
arm_model_error_t arm_model_read_config(uint32_t* data, uint32_t address, uint32_t
items);
```

3.8 Write data to configuration space

All transactions are 32-bits wide. A description of the related product registers and memory address map are provided with each product documentation set in the form of a register map.

- **data** - pointer to buffer for data to be written.
- **address** - start address aligned to 32-bit boundary.
- **items** - number of 32-bit words to read.

```
arm_model_error_t arm_model_write_config(const uint32_t* data, uint32_t address,
uint32_t items);
```

3.9 Read data from configuration space as strings

The configuration item is named according to the configuration hierarchy. See the section Script file format for more details.

- **name** - name of required items to read. For Chicken bits, the register names should be pre-fixed by the “$” symbol. For example: $Chicken_out_reg1

Note: You can use the `rr *` ***command to print the values of all the config space registers.***

- **value** - string to store the result
- **value_length** - length of value string.

```
arm_model_error_t arm_model_read_config_string(const char* name, char* value,
size_t value_length);
```

3.10 Write data to configuration space as strings

The configuration item is named according to the configuration hierarchy. See the section Script file format for more details.

- **name** - name of required items to write. For Chicken bits, the register names should be pre-fixed by the “$” symbol. For example: $Chicken_out_reg1
- **value** - expression to write.

```
arm_model_error_t arm_model_write_config_string(const char* name, const char*
value);
```

3.11 Process frames with current configuration

The number of input and output frames are predefined in `project_settings.h`. If there is no output, all frame dimensions are zero. The Caller must pre-allocate memory for output frames and specify the allocated memory size in the corresponding field.

- **frames_in** - input frames for processing.
- **frames_out** – pre-allocated storage for output frames.
- **dump** - pointer to array with modules dump (might be NULL).
- **dump_num** - number of modules for output dump.

```
arm_model_error_t arm_model_process_frames(const frame_t* frames_in, frame_t*
frames_out, dump_t* dump,
size_t dump_num);
```

3.12 Reset IP core

Reset all modules in the IP core to their initial state and set the configuration space to its default values. This clears internal memory as well.

```
arm_model_error_t arm_model_reset(void);
```

3.13 Typical usage

The library can be used in two ways:

1. Create the script file and call function `arm_model_process_script` – See the section Script file format for more details.
2. Use API functions as follows:
    a. Write register values by ether **`arm_model_write_config()`** or **`arm_model_write_config_string()`**.
    b. Fill frame in structure **frames_in**. Prepare memory for output frames in **frames_out**. Call function **`arm_model_process_frames()`**.
    c. If required read registers by ether **`arm_model_read_config()`** or **`arm_model_read_config_string()`**.

4 Executable command line parameters

The executable command line parameters used with the C model are as follows:

Command line parameters	Description
`-i scr_file_name`	name of script file to run (see Script file format for more information)
`-o read_out_file_name`	name of file where all read command store their values; if there are no read commands in the script file output file have a zero size after execution.
`-dump`	dump debug outputs from all modules in the model
`-trace`	show path of input frame coming through the model

5 Script file format

The file structure contains several units for processing. Each unit has configuration parameters for writing, and result configuration is applied to all frames in a unit. Every unit has one or more frames for processing. Also, a unit specifies configuration parameters for reading which are processed on a per-frame basis before a frame is processed. Finally, a unit can specify output files for the whole top level or specific modules. Each unit ends with a keyword process and is optionally followed by the number of the context. This section describes the detail of each element of a unit. See the section Examples for more information.

5.1 Comments

The script can contain comments denoted by the symbol #. In each line of code, everything that follows the # sign, including the # sign itself, is not processed. It is possible to add comments not only as individual lines but also after the active part of a string.

For example:

- # This is a comment line
- `indata random.frm` # this is an inline comment.

5.2 Configuration for writing

Configuration is based on the `config.xml` file which can include several other xml files. The file specification is as follows:

- The file specifies the name of registers and addresses.
- Fields in `config.xml` are not supported.
- The script can access registers by either, name with a keyword **wr**, or an address with a keyword **wa**.
- In name notation instance, module and register names are separated by a dot and represented in lowercase format.
- In address notation, all transactions are 32-bits long and addresses must be 4-bytes aligned.
- Address access can optionally specify a mask after the data value. If a bit = `1` in the mask, it means the data in this bit is overwritten but if a bit = `0`, it retains the original value.
- Address, data and mask values are always in hex format.

For example:

```
wr isp.top.active_width C00 # set value 0xC00 to register active_width in module
top
wa 400 C00 # write value 0x00000C00 by address 0x00000400
wa 420 CF0 FF00 # write value 0x00000C00 by address 0x00000400 with mask
0x0000FF00 (so only second
byte)
wr isp.top.rggb_start 1 # set value 0x1 to register rggb_start in module top with
mask 0xF
```

Some parameters are arrays. The `config.xml` file explicitly specifies the number and bit size of items. In name access notation script file can specify a zero-based index of an item or refer to the whole array by separating items with a comma. The C++ model shows a warning if the number of elements in an array or *Look Up Table* (LUT) does not correspond to the number of elements in a script. Indexes are in hexadecimal format.

For example:

```
wr isp.noise_profile.weight_lut[10] 1F # write 0x1F to item 17 (index 0x10) in
noise_profile.weight_lut
wr isp.noise_profile.weight_lut 10,20,10,0,65 # filling whole weight_lut (assuming
it has only 5
elements)
```

Accessing index-based lookup tables is easier in name format. An example string follows:

```
wr isp.iridix.asymmetry[8] 1F
```

is equal to

```
wa 800 8 # write index value
wa 804 1F FFFF # write value to LUT by index from address 0x800
```

Memory blocks like **metering_mem** are represented as a 32-bit array in name notation. The only difference is they usually have only module name, so the register name is omitted.

For example:

```
wr isp.metering_mem[123] DEADBEEF
wa 8123 DEADBEEF
```

5.3 Input data files

The input data files contain image frames, or set of frames. The file format is described in the Frame file format . The input data files are declared with the keyword **indata**. Each file has a specific format and can contain several frames with different resolutions. The example string that follows specifies reading all frames from the file:

```
indata random.frm
```

If a specific frame is required then the frame number started from 0 should be specified after the name of the file.

For example, if `random.frm` has 3 frames, the following strings read the first and third frame:

```
indata random.frm 0
indata random.frm 2
```

The script can specify several different input files and multiple frames for each file. All of them are processed based on settings as shown in the section Configuration for writing. The concept can be condensed into a two-step process as follows:

1. Configuration is applied.
2. All specified input frames are processed in order of description in the script file, that is, one by one.

5.4 Input Synchronization files

Synchronization files are used for *broken frame* simulation in HDL. The C++ model ignores this file, so the C++ model output files do not match the HDL model for *broken frame*. Synchronization files are used for all frames in a unit and these are specified by the keyword **syncdata**.

For example:

```
syncdata sync.txt
```

5.5 Configuration for reading

Modules can return data in the configuration area. The script can define reading fragments in both name (keyword **rr**) and address notations (**ra**). The name notation can omit register names which means, read all registers from the mentioned module in alphabetical order. Address read specifies the range of data for reading by its start and end address. For example:

```
rr isp.iridix.asymmetry[0] # read first item of Asymmetry LUT
rr isp.iridix.asymmetry # read whole Asymmetry LUT
rr isp.statistics # read all registers from statistics
ra 800 804 # read data from range 0x800 to 0x804 based on 32-bits transactions (so
actually
from 0x800
to 0x807)
rr isp.metering_mem # read whole metering memory
```

Reading can have a parameter to wait for an interrupt with a specific number. This value is ignored in the C++ model. For example:

```
rr isp.metering_mem 2 # read whole metering memory after interrupt 2 occurs
ra 800 800 2 # read data from 0x800
```

Configuration readings are saved in the output file on a per-frame basis. All reading commands are repeated after frame processing one by one in order of appearance in the script file. If the reading command must wait for the interrupt, it waits and won't go to the next reading command. This ensures that the output file from both the HDL and C++ model contain reading data in exactly same order.

In case of multiple input frames, read data for each frame are separated by the keyword **end_of_frame**.

5.6 Output files

Saving frames data to output files is optional. Verification can be based on the frame *Cyclic Redundancy Check* (CRC) which is implemented after each module. For CRC comparison, the script must specify the appropriate reading command. Frames after each module can be saved by the keyword **outdata** followed by the optional module name and a file name.

Memory transactions are saved with the keyword **memdata**. All frames in a unit are saved sequentially in one file, in the order as described in the section Input data files. If the output module is not specified, AD saves the whole model output. If there are several outputs like *Full Resolution* (FR) and *Down Scaler* (DS), it is saved in separate files with prefixes. For example:

```
outdata out.frm
outdata sinter sinter_out.frm
memdata fr_dma_writer fr_out.frm
memdata temper temper.frm
```

Note: *The output file can contain non-video data like AXI transactions on the Direct Memory Access (DMA) Writer.*

Two keywords can be used in output data names. These are described as follows:

- **num** – Auto number sequence the output file name, for example 0, 1, 2. The auto number sequence is only applied to unit **outdata** that has been set to {**num**}. Any unit **indata** or unit **outdata** set to a defined file name retains its name in the output data file.
- **indata** - This is substituted by the first input file name in a unit. For example:

```
indata input_data_first.frm
indata input_data_second.frm
outdata out_{indata}.frm
outdata out_{indata}_${num}.frm
process
```

In the previous example:

`out_{indata}` is replaced by `out_input_data_first.frm`, while outdata `out_{indata}_${num}.frm` is replaced as `out_input_data_first_0.frm`.

5.7 Including another script

One script file can call another script file by the keyword script. Calling another script means in place text substitution from the mentioned file.

For example:

```
script test1.scr
script test2.scr
script test3.scr
```

Each substituted file can contain another script files as well. This approach is useful for creating a global test which contains all small tests inside.

5.8 Writing data to model of DDR memory

Commands **wmh** and **wmb** writes data from a specified 32-bit text hex file or pure binary file to the specified *Double Data Rate* (DDR) memory address. The address must be 32-bits aligned.

For example:

```
wmh 1000 dump.txt # write data from hex file dump.txt to address 0x1000
wmb 2000 dump.bin # write data from binary file dump.bin to address 0x2000
```

5.9 Reading data from model of DDR memory

Commands **rmh** and **rmb** reads data from a specified memory address to hex or pure binary file. Second parameter is required size. The address must be 32-bits aligned.

For example:

```
rmh 1000 100 dump.txt # read 0x100 bytes of data from address 0x1000 to hex file
dump.txt
rmb 2000 100 dump.bin # read 0x100 bytes of data from address 0x2000 to binary
file dump.bin
```

6 Examples

6.1 Example 1

Read one frame, set `isp.top.total_width` to `0x100` and save data from FR DMA writer to file with automatic number.

```
# Unit 1. Frame HD
indata in_1280x720.frm
syncdata sync_1280x720.txt

wr isp.top.total_width 400

outdata fr_dma_writer out_fr_dma_writer_${num}.frm

rd isp.checksum.fr_crop

process

# Unit 2. Frame full HD
indata in_1920x1080.frm
syncdata sync_1920x1080.txt

wr isp.top.total_width 500

outdata fr_dma_writer out_fr_dma_writer_${num}.frm

rd isp.checksum.fr_crop

process
```

6.2 Example 2

Read several frames and save all results in one file.

```
# Multiple frames
indata in_1280x720.frm 0
indata in_1920x1080.frm 5
indata random.frm
syncdata sync.txt

outdata out.frm
memdata mem.trn

process 1 # process data in context 1
script test1.scr # run another script
```

7 Frame file

This section provides details about the frame file.

7.1 Frame file Structure

The .frm file consists of one or more portions. Each portion always contains a header with meta-information about the data, and the data itself.

The data contains samples arranged in two-dimensional layers. Within a given portion, each layer has same dimensions.

The header should be interpreted as ASCII text (one byte per symbol). The header must start with a comment string which contains the format identifier `#ISP1` followed by dot symbol `.` and five names of fields delimited with a dot symbol `.`.

The length of comment string is always 37. The size of header is 64 bytes, if the new line is `"\n"(ASCII 0A.`

7.2 Example of comments string

```
#ISP2.width.height.depth.type.layers:
```

The comment string and each field are delimited with a newline. Newline is always `"\n"(ASCII 0A)` irrespective of the platform.

If you use 2 bytes as newline `"\r\n" (ASCII 0D 0A)`, the framework does read it properly but be aware that standard utility functions such as `md5sum` and `diff` might report that there are differences in the file.

The names of fields are as follows:

Code	Description
width	width (length of a line) of layer in pixels (samples)
height	height (number of lines in layer) in pixels (samples)
depth	number of bits carrying data in each sample
type	total number of bits in each sample (data alignment)
layers	number of layers in portion

The order of names inside the comment string is not fixed, but they must all be present, the order must correspond to fields position in the remaining part of the header. The following

table specifies sizes of each part of header and valid values for each of them and the order of the fields should be preserved:

Code	Format	Possible valid values
comment	37 bytes + newline	
width	7 bytes + newline	Decimal integer 1 to 99999 (inclusive)
height	7 bytes + newline	Decimal integer 1 to 99999 (inclusive)
depth	2 bytes + newline	Decimal integer 1 to 64 (inclusive)
type	2 bytes + newline	Only these values: 8, 16, 32, 64
layers	3 bytes + newline	Only these values: 001, 003,

The value of the field `depth` must be less than or equal to value of field `type`.

Each integer value must be left-padded with zeros.

7.3 Example of header

3 layers of 12 bit data with resolution 1920 x 1080 aligned by 16 bits:

```
#ISP1.width.height.depth.type.layers
  0001920
  0001080
  12
  16
  003
```

Data begins with the 64-th byte (Counting from 0). The size of data part (in bytes) must be:

```
width x height x layers x type / 8
```

Total size of each portion (in bytes) is:

```
64 + width x height x layers x type / 8
```

Next portion, if any, starts immediately after previous portion.

7.4 Data Encoding

Data part of each portion of an `.frm` file must be encoded according to header information. Each sample (pixel) is stored as an unsigned integer of 1, 2, 4, or 8 bytes for `type` values 8, 16, 32, and 64 correspondingly. The byte order is always little-endian. Bit alignment is always LSB, which means that lowest `depth` bits are used out of total available `type` bits (remaining MSB are zeros).

Samples are arranged in lines (of length `width`). A `height` number of lines resemble a layer. Total number of layers is `layers`. There are no separators or delimiters neither between lines nor between layers.