EDU-COMPRESSOR: Educational Streaming Compressor PCI Device

Technical Reference Manual

Virtual Hardware Division

September 2026

Contents

0.1 Document Information

Item	Value
Product Name	EDU-COMPRESSOR
Device ID	0xBEEF
Vendor ID	0x1234
Revision	0x01
Interface	Conventional PCI
Function	Streaming data compression
Compression Algorithm	zlib/DEFLATE
Operation Mode	Chunked streaming
Interrupt Support	PCI INTx
Input Buffer	32 KiB

Item	Value
Output Buffer	32 KiB

1 Overview

The EDU-COMPRESSOR device is a memory-mapped PCI streaming compression device intended for device driver and operating system education.

The device implements a simple hardware-style interface to a zlib compression stream. Software provides input data through a fixed-size input buffer and retrieves compressed data from a fixed-size output buffer. Although the device exposes fixed 32 KiB input and output windows, compression is performed as a streaming operation. A single compression stream may therefore process files significantly larger than either buffer. The device uses an explicit command and state model. Software starts a compression stream, repeatedly submits input chunks and consumes output chunks, and finally finishes the stream. Completion and buffer events may optionally generate PCI INTx interrupts.

2 Features

- Conventional PCI device
- Memory-mapped I/O (MMIO) interface
- zlib/DEFLATE compression
- Streaming compression model
- Fixed 32 KiB input buffer
- Fixed 32 KiB output buffer
- Configurable compression level
- Input and output byte counters
- Explicit stream start and finish operations
- Interrupt-driven operation
- Polling-compatible status interface
- Write-one-to-clear interrupt acknowledgement
- Error reporting
- No DMA support, no internal request queue

3 PCI Configuration

3.1 Device Identification

Field	Value
Vendor ID	0x1234
Device ID	0xBEEF
Revision ID	0x01
Class Code	Other Device
Interrupt Pin	INTA

4 Memory Map

The device exposes a single memory-mapped Base Address Register (BAR).

4.1 BAR0

Property	Value
BAR Number	0
Size	0x20000 bytes (128 KiB)
Type	Memory-mapped

4.2 BAR0 Layout

Address Range	Description
0x0000–0x0037	Control and status registers
0x0038–0x0FFF	Reserved
0x1000–0x8FFF	Input buffer (32 KiB)
0x9000–0x10FFF	Output buffer (32 KiB)
0x11000–0x1FFFF	Reserved

5 Register Summary

Unless otherwise stated, control and status registers are 32-bit little-endian values.

Buffer memory supports accesses between 1 and 4 bytes.

Offset	Register	Size	Access
0x000	ID	32 bits	RO
0x004	VERSION	32 bits	RO
0x008	COMMAND	32 bits	WO
0x00C	STATUS	32 bits	RO
0x010	INPUT_LENGTH	32 bits	RW
0x014	OUTPUT_LENGTH	32 bits	RO
0x018	LEVEL	32 bits	RW
0x01C	IRQ_STATUS	32 bits	RO
0x020	IRQ_ENABLE	32 bits	RW
0x024	IRQ_ACK	32 bits	WO
0x028	TOTAL_INPUT	32 bits	RO
0x02C	TOTAL_OUTPUT	32 bits	RO
0x030	BUFFER_SIZE	32 bits	RO
0x034	ERROR	32 bits	RO

Access legend: RO = Read-only, WO = Write-only, RW = Read-write.

6 Buffer Memory

The device contains two fixed-size memory windows.

6.1 Input Buffer

Base address: `0x1000`

Size: 32768 bytes (32 KiB)

Software writes uncompressed input data to this buffer before issuing the SUBMIT command. The number of valid bytes in the buffer must be specified through the INPUT_LENGTH register.

Example:

```
memcpy_toio(bar0 + EDU_COMP_INPUT_BASE, input_data, input_length);
writel(input_length, bar0 + REG_INPUT_LENGTH);
writel(CMD_SUBMIT, bar0 + REG_COMMAND);
```

6.1.1 Behaviour

- The input buffer is writable while no submitted input is being processed.
- The maximum input chunk size is 32768 bytes.
- INPUT_LENGTH must be greater than zero and no greater than BUFFER_SIZE.
- The driver must not modify the input buffer while STATUS.INPUT_BUSY is set.
- Input data is consumed by the active compression stream.
- Once all submitted input has been consumed, an INPUT_CONSUMED interrupt may be generated.

6.2 Output Buffer

Base address: `0x9000`

Size: 32768 bytes (32 KiB)

Compressed output is made available in this buffer. The number of valid bytes is reported through OUTPUT_LENGTH.

Example:

```
uint32_t output_length;

output_length = readl(bar0 + REG_OUTPUT_LENGTH);
memcpy_fromio(output_data, bar0 + EDU_COMP_OUTPUT_BASE, output_length);
```

After copying the output data, software must notify the device by issuing the OUTPUT_CONSUMED command.

```
writel(CMD_OUTPUT_CONSUMED, bar0 + REG_COMMAND);
```

6.2.1 Behaviour

- Output is valid from offset zero up to OUTPUT_LENGTH.
- Reading beyond OUTPUT_LENGTH returns zero.
- A new input submission cannot be accepted while output remains pending.
- Software must consume output before the device can continue producing additional output.
- OUTPUT_CONSUMED clears the current output buffer and allows compression to continue.

7 Register Descriptions

7.1 ID Register

Offset: `0x000`

Access: Read-only

Bits	Name	Description
31:0	IDENTIFIER	Device identification value

The ID register contains:

```
0x4544434D
```

Interpreted as ASCII characters, this corresponds to:

```
"EDCM"
```

Example:

```
uint32_t id = readl(bar0 + REG_ID);
```

8 VERSION Register

Offset: `0x004`

Access: Read-only

Bits	Name	Description
31:0	VERSION	Device version

The current version is:

```
0x00010000
```

This represents version 1.0 of the device interface.

9 COMMAND Register

Offset: `0x008`

Access: Write-only

The COMMAND register controls the compression stream.

Only one command bit may be written at a time. Writing a value containing multiple command bits is considered an invalid command.

Bit	Name	Description
0	RESET	Reset device state
1	START	Start a compression stream
2	SUBMIT	Submit the current input buffer
3	OUTPUT_CONSUMED	Release the current output buffer
4	FINISH	Finish the compression stream
31:5	Reserved	Invalid

9.1 RESET Command

```
writel(CMD_RESET, bar0 + REG_COMMAND);
```

RESET terminates any active compression stream and restores the device to its initial state. The following state is reset:

- Active compression stream
- Input length
- Output length
- Compression level
- Interrupt status
- Interrupt enable mask
- Total input counter
- Total output counter
- Error code
- DONE state
- Input buffer contents
- Output buffer contents

The default compression level after RESET is the zlib default compression level (6).

9.2 START Command

```
writel(CMD_START, bar0 + REG_COMMAND);
```

START creates a new compression stream. A compression level may be configured through LEVEL before issuing START.

When successful:

- A new zlib compression stream is created.
- STATUS.ACTIVE is set.
- Total input and output counters are reset.
- Previous error status is cleared.

START is invalid if:

- A compression stream is already active.
- Submitted input is still being processed.
- Output data is pending.

An invalid START command sets an error condition.

9.3 SUBMIT Command

```
writel(CMD_SUBMIT, bar0 + REG_COMMAND);
```

SUBMIT provides the current contents of the input buffer to the compression stream.

Before issuing SUBMIT, software must:

1. Copy input data to the input buffer.
2. Set INPUT_LENGTH to the number of valid bytes.
3. Ensure that no output is currently pending.

Example:

```
memcpy_toio(bar0 + EDU_COMP_INPUT_BASE, buffer, length);
writel(length, bar0 + REG_INPUT_LENGTH);
writel(CMD_SUBMIT, bar0 + REG_COMMAND);
```

SUBMIT is invalid if:

- No compression stream is active.

- The device is finishing.
- Previous input is still being processed.
- Output data is pending.
- INPUT_LENGTH is zero.
- INPUT_LENGTH exceeds BUFFER_SIZE.

When accepted, STATUS.INPUT_BUSY is set while submitted input remains to be processed.

9.4 OUTPUT_CONSUMED Command

```
writel(CMD_OUTPUT_CONSUMED, bar0 + REG_COMMAND);
```

This command informs the device that software has copied the current compressed output.

The device clears OUTPUT_LENGTH and may resume compression if additional input or finishing work remains. OUTPUT_CONSUMED is valid only when output data is pending.

The driver should use the following sequence:

```
length = readl(bar0 + REG_OUTPUT_LENGTH);
memcpy_fromio(destination, bar0 + EDU_COMP_OUTPUT_BASE, length);
writel(CMD_OUTPUT_CONSUMED, bar0 + REG_COMMAND);
```

9.5 FINISH Command

```
writel(CMD_FINISH, bar0 + REG_COMMAND);
```

FINISH completes the current compression stream. The command causes zlib to flush remaining compression state and generate the final bytes required to complete the compressed stream. No additional input may be submitted after FINISH.

FINISH is invalid if:

- No compression stream is active.
- Input data is still being processed.
- Output data is pending.
- The device is already finishing.

Compression may produce one or more output buffers while finishing. Software must continue consuming output until STATUS.DONE is set.

10 STATUS Register

Offset: `0x00C`

Access: Read-only

Bit	Name	Description
0	READY	Device is operational
1	ACTIVE	Compression stream is active
2	INPUT_BUSY	Submitted input is being processed
3	OUTPUT_READY	Compressed output is available
4	FINISHING	Compression stream is being finished
5	DONE	Compression stream completed
6	ERROR	Device error occurred
31:7	Reserved	Reads as zero

10.1 READY Bit

The READY bit indicates that the device is operational.

```
0 = Device unavailable
1 = Device operational
```

The current implementation reports READY whenever the device is operational, including while a compression stream is active.

10.2 ACTIVE Bit

The ACTIVE bit indicates that a compression stream exists.

```
0 = No active compression stream
1 = Compression stream active
```

ACTIVE is set by START. It is cleared after successful completion of FINISH or after RESET.

10.3 INPUT_BUSY Bit

INPUT_BUSY indicates that submitted input has not yet been completely consumed.

```
0 = No submitted input pending
1 = Input is being processed
```

While INPUT_BUSY is set, software must not overwrite the input buffer.

10.4 OUTPUT_READY Bit

OUTPUT_READY indicates that compressed data is available in the output buffer.

```
0 = No output pending
1 = Output data available
```

When OUTPUT_READY is set:

- OUTPUT_LENGTH specifies the number of valid bytes.
- Software should copy the data from the output buffer.
- Software must issue OUTPUT_CONSUMED before additional output can be generated.

10.5 FINISHING Bit

FINISHING indicates that the device is completing the compression stream.

```
0 = Normal compression operation
1 = Final stream output is being generated
```

FINISHING remains set until the device reaches the end of the compression stream.

10.6 DONE Bit

DONE indicates that the compression stream has completed successfully.

```
0 = Stream not complete
1 = Stream complete
```

When DONE is set:

- The compression stream has been terminated.
- No further input may be submitted.
- TOTAL_INPUT and TOTAL_OUTPUT contain final counters.
- An IRQ_DONE interrupt cause is raised.

10.7 ERROR Bit

ERROR indicates that the device has encountered an error.

```
0 = No error
1 = Error condition present
```

The detailed error code is available through the ERROR register.

11 INPUT_LENGTH Register

Offset: `0x010`

Access: Read-write

Bits	Name	Description
31:0	LENGTH	Number of valid input bytes

INPUT_LENGTH specifies the number of valid bytes currently stored in the input buffer.

The value must satisfy:

```
1 <= INPUT_LENGTH <= BUFFER_SIZE
```

Example:

```
writel(input_length, bar0 + REG_INPUT_LENGTH);
```

11.0.1 Behaviour

- INPUT_LENGTH must be configured before SUBMIT.
- Writing a value greater than BUFFER_SIZE causes an error.
- Writing while INPUT_BUSY is set causes an error.
- INPUT_LENGTH is cleared when the submitted input has been fully consumed.

12 OUTPUT_LENGTH Register

Offset: `0x014`

Access: Read-only

Bits	Name	Description
31:0	LENGTH	Number of valid output bytes

OUTPUT_LENGTH reports the number of valid compressed bytes currently available in the output buffer.

Example:

```
uint32_t length;
length = readl(bar0 + REG_OUTPUT_LENGTH);
```

A non-zero value corresponds to STATUS.OUTPUT_READY.

OUTPUT_LENGTH is cleared when software issues OUTPUT_CONSUMED.

13 LEVEL Register

Offset: `0x018`

Access: Read-write

Bits	Name	Description
31:0	LEVEL	Compression level

LEVEL selects the zlib compression level used when START creates a new compression stream.

Valid values are 0 through 9:

Level	Description
0	No compression
1	Fast compression
6	Typical balanced compression
9	Maximum compression

Upon initialisation, the device initialises sets LEVEL to 6.

Example:

```
writel(8, bar0 + REG_LEVEL);
```

LEVEL may only be modified while no compression stream is active. Attempting to change LEVEL while ACTIVE causes an error.

14 IRQ_STATUS Register

Offset: `0x01C`

Access: Read-only

IRQ_STATUS records pending interrupt causes.

Bit	Name	Description
0	INPUT_CONSUMED	Submitted input was consumed
1	OUTPUT_READY	Compressed output is available
2	DONE	Compression stream completed
3	ERROR	Device error occurred
31:4	Reserved	Reads as zero

A PCI INTx interrupt is asserted when:

```
IRQ_STATUS & IRQ_ENABLE != 0
```

14.1 INPUT_CONSUMED

This bit is raised when all bytes from a submitted input buffer have been consumed.

When this interrupt occurs, the driver may prepare the next input chunk.

14.2 OUTPUT_READY

This bit is raised when compressed output is available.

The driver should:

1. Read OUTPUT_LENGTH.
2. Copy the compressed bytes from the output buffer.
3. Issue OUTPUT_CONSUMED.

14.3 DONE

This bit is raised when FINISH successfully completes the compression stream.

DONE may be raised together with OUTPUT_READY if the final compressed bytes are present in the output buffer.

14.4 ERROR

This bit is raised when the device enters an error condition.

The specific error can be read from the ERROR register.

15 IRQ_ENABLE Register

Offset: `0x020`

Access: Read-write

IRQ_ENABLE controls which interrupt causes are allowed to assert the PCI interrupt.

Bit	Name	Description
0	INPUT_CONSUMED_ENABLE	Enable input consumed interrupts
1	OUTPUT_READY_ENABLE	Enable output ready interrupts
2	DONE_ENABLE	Enable completion interrupts
3	ERROR_ENABLE	Enable error interrupts
31:4	Reserved	Ignored

Example:

```
writel(IRQ_INPUT_CONSUMED | IRQ_OUTPUT_READY | IRQ_DONE | IRQ_ERROR,
  bar0 + REG_IRQ_ENABLE);
```

Enabling an interrupt while its cause is already pending may immediately assert the PCI interrupt.

16 IRQ_ACK Register

Offset: `0x024`

Access: Write-only

IRQ_ACK acknowledges interrupt causes using write-one-to-clear semantics.

Bit	Name	Description
0	INPUT_CONSUMED	Acknowledge input consumed
1	OUTPUT_READY	Acknowledge output ready
2	DONE	Acknowledge completion
3	ERROR	Acknowledge error
31:4	Reserved	Ignored

Example:

```
writel(IRQ_OUTPUT_READY, bar0 + REG_IRQ_ACK);
```

16.0.1 Behaviour

Writing a one clears the corresponding interrupt status bit. Writing a zero leaves the bit unchanged.

Some interrupt causes are level-related and may be reasserted while their underlying condition remains true. In particular:

- OUTPUT_READY remains asserted while output data remains pending.
- DONE remains asserted while the device remains in the DONE state.

Software must therefore resolve the underlying condition as well as acknowledge the interrupt.

17 TOTAL_INPUT Register

Offset: `0x028`

Access: Read-only

Bits	Name	Description
31:0	TOTAL	Total input bytes consumed

TOTAL_INPUT reports the cumulative number of uncompressed bytes consumed by the current compression stream. The counter is reset when START begins a new stream.

Example:

```
uint32_t total = readl(bar0 + REG_TOTAL_INPUT);
```

18 TOTAL_OUTPUT Register

Offset: `0x02C`

Access: Read-only

Bits	Name	Description
31:0	TOTAL	Total compressed bytes produced

TOTAL_OUTPUT reports the cumulative number of compressed bytes produced by the current compression stream. The counter includes all output chunks produced during the stream. The counter is reset when START begins a new stream.

19 BUFFER_SIZE Register

Offset: `0x030`

Access: Read-only

Bits	Name	Description
31:0	SIZE	Size of each streaming buffer

The register reports `32768` bytes. Both the input and output buffers have this size.

20 ERROR Register

Offset: `0x034`

Access: Read-only

The ERROR register provides a detailed error code.

Value	Name	Description
0	ERR_NONE	No error

Value	Name	Description
1	ERR_BAD_COMMAND	Invalid command
2	ERR_BAD_LENGTH	Invalid buffer length
3	ERR_BAD_STATE	Command invalid in current state
4	ERR_ZLIB	Compression library error

20.1 ERR_BAD_COMMAND

Generated when software writes an unsupported or malformed command value.

20.2 ERR_BAD_LENGTH

Generated when:

- INPUT_LENGTH is zero during SUBMIT.
- INPUT_LENGTH exceeds BUFFER_SIZE.
- INPUT_LENGTH is modified while input is busy.

20.3 ERR_BAD_STATE

Generated when an operation is attempted in an invalid state. Examples include:

- START while a stream is already active.
- SUBMIT without START.
- SUBMIT while output is pending.
- FINISH while input is still being processed.
- OUTPUT_CONSUMED when no output exists.
- Changing LEVEL while a stream is active.

20.4 ERR_ZLIB

Generated when the device’s underlying zlib compression implementation reports an unexpected error.

21 Programming Model

The device implements a streaming producer-consumer model. A complete compression operation consists of:

1. Reset the device if necessary.
2. Configure the compression level.
3. Optionally enable interrupts.
4. Issue START.
5. Copy an input chunk into the input buffer.
6. Set INPUT_LENGTH.
7. Issue SUBMIT.
8. Wait for input consumption or output availability.
9. Consume output when OUTPUT_READY is set.
10. Repeat steps 5-9 for additional input chunks.
11. Issue FINISH after all input has been submitted.
12. Continue consuming output until DONE is set.

22 Basic Compression Sequence

The following pseudocode demonstrates a polling-based compression sequence of a small amount of data (<= 32 KiB):

```
// Configure compression level.
writel(6, bar0 + REG_LEVEL);

// Start a new compression stream.
writel(CMD_START, bar0 + REG_COMMAND);

while (more_input) {

    // Copy at most 32 KiB into the input buffer.
    size_t length = get_next_chunk(input, 32768);
    memcpy_toio(bar0 + EDU_COMP_INPUT_BASE, input, length);
    writel(length, bar0 + REG_INPUT_LENGTH);
    writel(CMD_SUBMIT, bar0 + REG_COMMAND);

    // Process all output generated while consuming this input.
    while (readl(bar0 + REG_STATUS) & STATUS_OUTPUT_READY) {

        uint32_t output_length = readl(bar0 + REG_OUTPUT_LENGTH);
        memcpy_fromio(output, bar0 + EDU_COMP_OUTPUT_BASE, output_length);
        writel(CMD_OUTPUT_CONSUMED, bar0 + REG_COMMAND);
    }
}

// Finish the compression stream.
writel(CMD_FINISH, bar0 + REG_COMMAND);
```

```
// Continue consuming output until compression completes.
while (!(readl(bar0 + REG_STATUS) & STATUS_DONE)) {

    if (readl(bar0 + REG_STATUS) & STATUS_OUTPUT_READY) {

        uint32_t output_length = readl(bar0 + REG_OUTPUT_LENGTH);
        memcpy_fromio(output, bar0 + EDU_COMP_OUTPUT_BASE, output_length);
        writel(CMD_OUTPUT_CONSUMED, bar0 + REG_COMMAND);
    }
}
```

23 Interrupt Model

The device uses a conventional PCI INTx interrupt.

The interrupt line is asserted when at least one enabled interrupt cause is pending:

```
(IRQ_STATUS & IRQ_ENABLE) != 0
```

23.1 Typical Interrupt-Driven Sequence

A driver may configure interrupts as follows:

```
writel(IRQ_INPUT_CONSUMED | IRQ_OUTPUT_READY | IRQ_DONE | IRQ_ERROR, bar0 +
 ↪  REG_IRQ_ENABLE);
```

The interrupt handler can then inspect the pending causes:

```
uint32_t irq_status = readl(bar0 + REG_IRQ_STATUS);

if (irq_status & IRQ_OUTPUT_READY) {
    // Output is available.
}

if (irq_status & IRQ_INPUT_CONSUMED) {
    // Another input buffer may be submitted.
}

if (irq_status & IRQ_DONE) {
    // Compression stream completed.
}
```

```
if (irq_status & IRQ_ERROR) {
    // Inspect REG_ERROR.
}

// Acknowledge interrupt causes.
writel(irq_status, bar0 + REG_IRQ_ACK);
```

The driver must account for level-related interrupt conditions. For example, acknowledging OUTPUT_READY without issuing OUTPUT_CONSUMED may cause OUTPUT_READY to remain pending because compressed data is still available.

24 State Transitions

The device implements a command-driven state machine.

24.1 Initial State

After device initialisation or RESET:

```
READY = 1
ACTIVE = 0
INPUT_BUSY = 0
OUTPUT_READY = 0
FINISHING = 0
DONE = 0
ERROR = 0
```

The device is ready for configuration and START.

24.2 Idle State

```
ACTIVE = 0
INPUT_BUSY = 0
OUTPUT_READY = 0
```

Valid operations include:

- RESET
- LEVEL configuration
- START

24.3 Active State

After START:

```
ACTIVE = 1
INPUT_BUSY = 0
OUTPUT_READY = 0
FINISHING = 0
DONE = 0
```

The device accepts input submissions.

24.4 Input Processing State

After SUBMIT:

```
ACTIVE = 1
INPUT_BUSY = 1
```

The compression engine processes the submitted input.

Possible outcomes include:

- Input is consumed.
- Output becomes available.
- An error occurs.

24.5 Output Pending State

When compressed data has been generated:

```
OUTPUT_READY = 1
OUTPUT_LENGTH > 0
```

The driver must:

1. Copy the output.
2. Issue OUTPUT_CONSUMED.

The device may then continue processing remaining input or finishing work.

24.6 Finishing State

After FINISH:

```
ACTIVE = 1
FINISHING = 1
```

The device flushes the remaining compression stream. Multiple output buffers may be generated. The driver must continue consuming output until DONE is reached.

24.7 Done State

After the compression stream completes:

```
ACTIVE = 0
DONE = 1
FINISHING = 0
```

The final output buffer may still need to be consumed. A new stream may subsequently be started when the device is in an appropriate state.

24.8 Error State

When an invalid operation or compression failure occurs:

```
ERROR = 1
IRQ_ERROR pending
```

The detailed reason is available through the ERROR register. RESET can be used to restore the device to its initial state.

25 State Diagram

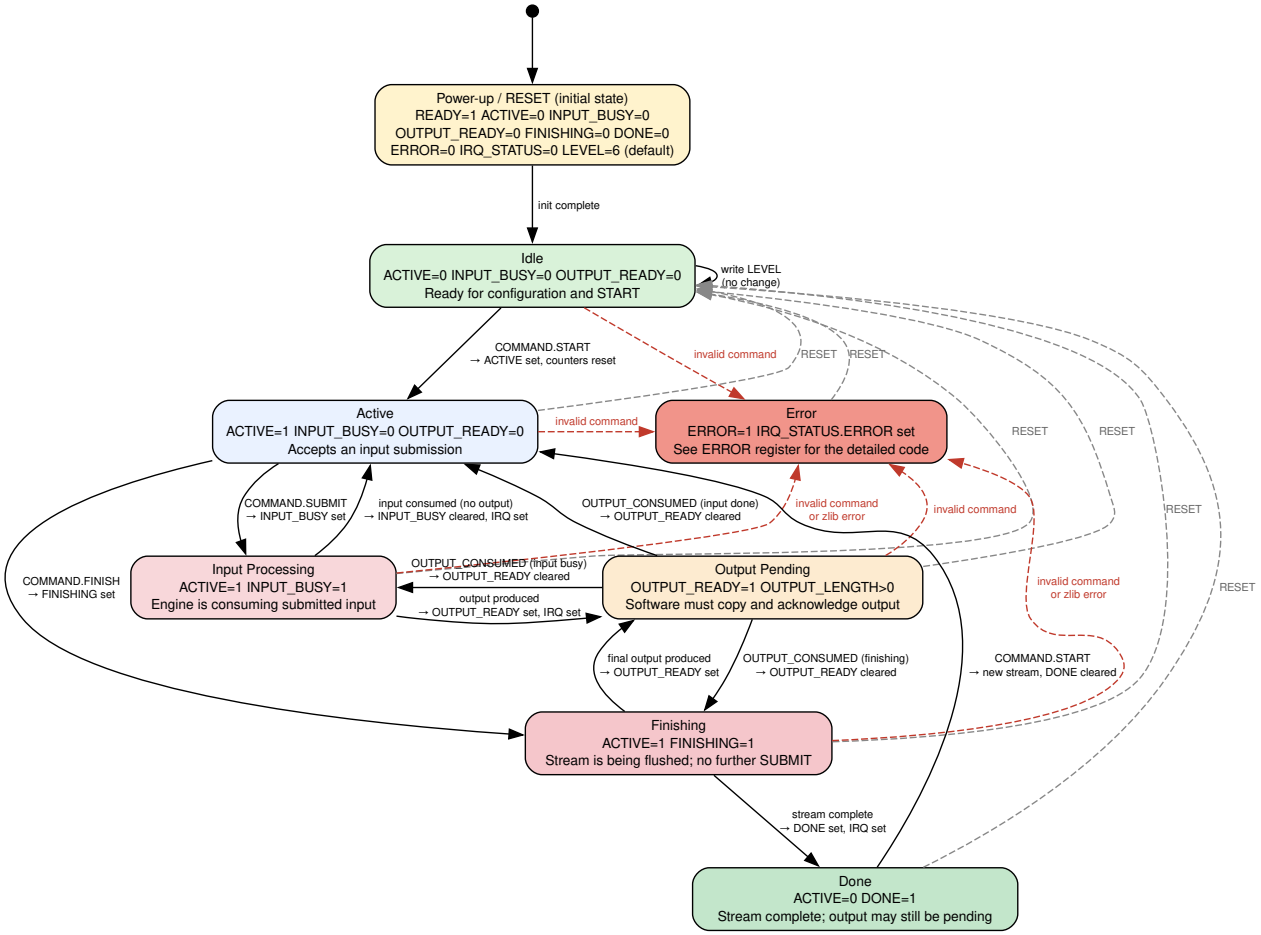

Figure 1: EDU-COMPRESSOR state transition graph

26 Device Behaviour

26.1 Reset Behaviour

RESET immediately terminates any active compression stream. The device:

- Clears all counters.
- Clears interrupt status.
- Disables interrupts.
- Clears pending output.
- Clears errors.
- Clears DONE.
- Restores the default compression level.
- Clears both input and output buffers.

The PCI interrupt is deasserted.

26.2 Input Consumption

A submitted input buffer may not necessarily produce output immediately. Therefore, INPUT_CONSUMED and OUTPUT_READY are independent events. A driver must not assume that consuming input always produces a corresponding output buffer.

26.3 Output Production

Output may be produced:

- During normal input processing.
- While finishing the compression stream.
- As the final bytes of the compressed stream.

When the output buffer becomes full or contains available data, compression pauses until software consumes the output.

27 Error Handling

Drivers should check STATUS.ERROR or handle IRQ_ERROR. The recommended error recovery sequence is:

1. Detect an error by reading STATUS.ERROR.
2. Determine the error type by reading the ERROR register.
3. Record or report the error.
4. Reset the device to its initial state with CMD_RESET.

Example:

```
uint32_t error;

if (readl(bar0 + REG_STATUS) & STATUS_ERROR) {
    error = readl(bar0 + REG_ERROR);
    printk(KERN_ERR "EDU-COMPRESSOR error %u\n", error);

    writel(CMD_RESET, bar0 + REG_COMMAND);
}
```

28 Polling Model

The device can be operated without interrupts by repeatedly reading STATUS.

Example:

```
for (;;) {
    uint32_t status = readl(bar0 + REG_STATUS);

    if (status & STATUS_ERROR) {
        /* Handle error. */
        break;
    }

    if (status & STATUS_OUTPUT_READY) {
        /* Consume output. */
    }

    if (status & STATUS_DONE)
        break;

    cpu_relax();
}
```

Polling is simpler but less efficient than interrupt-driven operation.

29 Interrupt-Driven Model

An interrupt-driven driver can sleep while waiting for device events.

A typical sequence is:

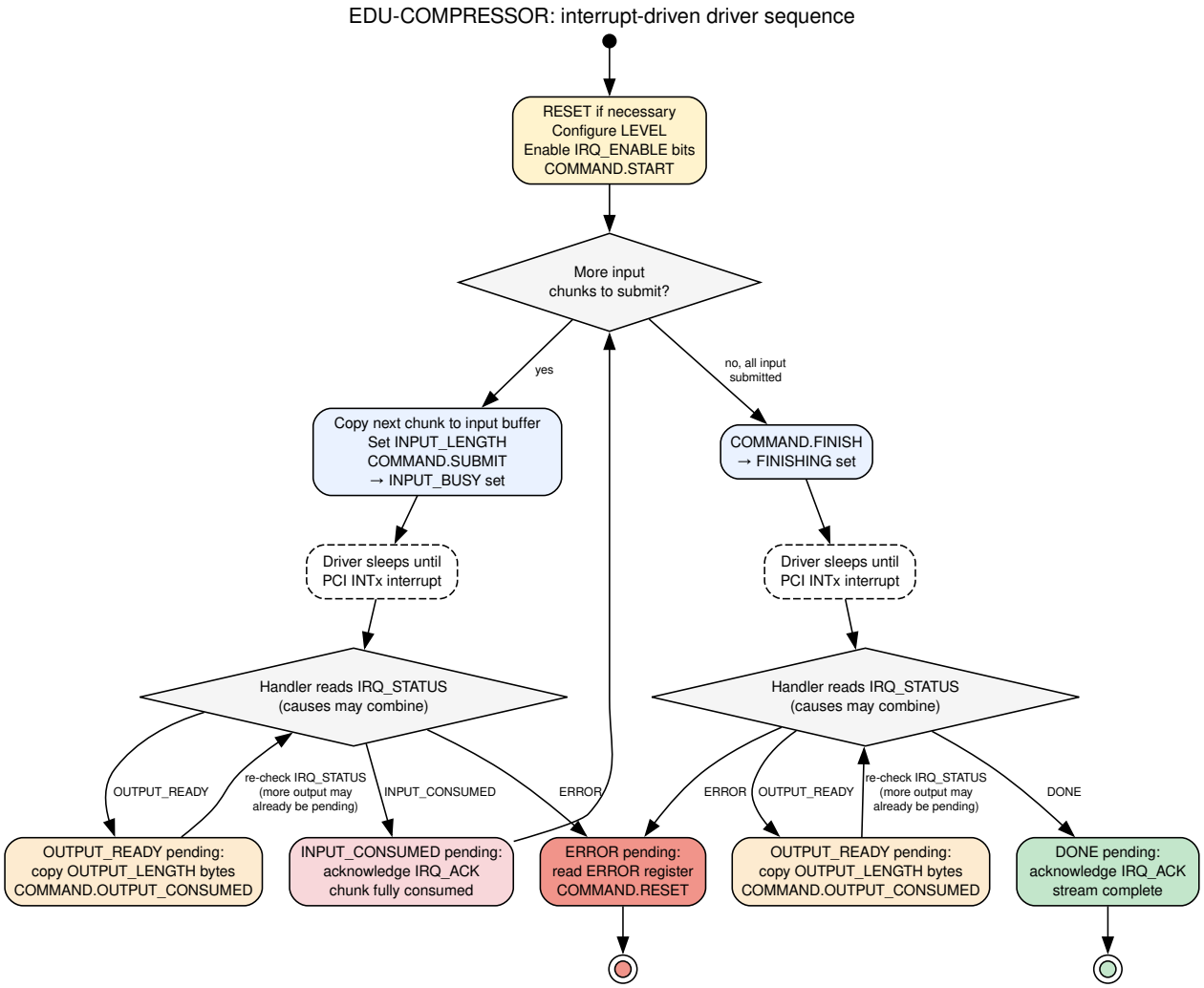

Figure 2: EDU-COMPRESSOR interrupt-driven driver sequence

A single SUBMIT may yield zero, one, or several OUTPUT_READY interrupts before the matching INPUT_CONSUMED interrupt arrives, since the two causes are independent and may also be pending together. The same applies while FINISHING: several OUTPUT_READY interrupts can precede the final DONE interrupt.

30 Appendix A. Device Constants

```
#define EDU_COMP_VENDOR_ID      0x1234
#define EDU_COMP_DEVICE_ID      0xbeef
#define EDU_COMP_REVISION       0x01

#define EDU_COMP_IDENT          0x4544434dU
```

```
#define EDU_COMP_VERSION        0x00010000U

#define EDU_COMP_BUFFER_SIZE    (32 * 1024)

#define EDU_COMP_BAR_SIZE       0x20000

#define EDU_COMP_INPUT_BASE     0x01000
#define EDU_COMP_OUTPUT_BASE    0x09000
```

31 Appendix B. Register Offsets

```
#define REG_ID                  0x000
#define REG_VERSION             0x004
#define REG_COMMAND             0x008
#define REG_STATUS              0x00c
#define REG_INPUT_LENGTH        0x010
#define REG_OUTPUT_LENGTH       0x014
#define REG_LEVEL               0x018
#define REG_IRQ_STATUS          0x01c
#define REG_IRQ_ENABLE          0x020
#define REG_IRQ_ACK             0x024
#define REG_TOTAL_INPUT         0x028
#define REG_TOTAL_OUTPUT        0x02c
#define REG_BUFFER_SIZE         0x030
#define REG_ERROR               0x034
```

32 Appendix C. Command Definitions

```
#define CMD_RESET               BIT(0) // (1U << 0)
#define CMD_START               BIT(1) // (1U << 1)
#define CMD_SUBMIT              BIT(2) // (1U << 2)
#define CMD_OUTPUT_CONSUMED     BIT(3) // etc.
#define CMD_FINISH              BIT(4)
```

33 Appendix D. Status Definitions

```
#define STATUS_READY            BIT(0)
#define STATUS_ACTIVE           BIT(1)
#define STATUS_INPUT_BUSY       BIT(2)
#define STATUS_OUTPUT_READY     BIT(3)
#define STATUS_FINISHING        BIT(4)
#define STATUS_DONE             BIT(5)
#define STATUS_ERROR            BIT(6)
```

34 Appendix E. Interrupt Definitions

```
#define IRQ_INPUT_CONSUMED      BIT(0)
#define IRQ_OUTPUT_READY        BIT(1)
#define IRQ_DONE                BIT(2)
#define IRQ_ERROR               BIT(3)

#define IRQ_ALL (IRQ_INPUT_CONSUMED | IRQ_OUTPUT_READY | IRQ_DONE | IRQ_ERROR)
```

35 Appendix F. Error Codes

```
enum EduCompressorError {
    ERR_NONE        = 0,
    ERR_BAD_COMMAND = 1,
    ERR_BAD_LENGTH  = 2,
    ERR_BAD_STATE   = 3,
    ERR_ZLIB        = 4,
};
```