ABBYCUS IOX  ABBY-IOX-UART

A low cost - high performance I/O co-processor module performs I/O expansion and performance enhancements that can significantly enhance the capabilities of Microprocessor system designs.

Features

- High speed serial UART bus performs serial to parallel I/O expansion with simple integration to MPU's like ESP32, RP2040, Raspberry Pi, STM32Fxxx, etc. *Other serial bus options are available, see below.*
- Up to 20 independent GPIO pins, independently configurable as input or output.
- Outputs can be set as push-pull or open drain type with programmable current drive.
- Pull-up and pull-down resistors can be applied to any I/O pin (inputs ***or*** outputs).
- Up to 16 inputs can be configured to generate input events on rising, falling, or both edges.
- Events can trigger a programmable output pin (hardware notification). Also can be read by software.
- Up to 10 outputs can be configured for PWM output.
- Up to 10 inputs can be used as digital capture windows (frequency and pulse width measurements). Capture event can trigger an event output pin (hardware notification).
- 12 bit Analog To Digital Converter (ADC) with up to 10 multiplexed inputs. Range = 0 - 3.3 VDC typical. ADC can generate event output on conversion complete.
- Support for 8 rotary encoders with direction, count, and rotational speed calculated. Encoder motion can trigger an event output.
- UART baud rate can be changed to any value up to 6 Mbaud.
- Digital & I/O supply voltage operation from 2.5V to 3.6V.
- Low power operation with sleep and wake from sleep.
- Small 1.6cm square module can be mounted via 1.27mm (0.05 inch) header, PCB surface mounted, or DIY wired.
- Module contains STM32F 32-bit MCU with crystal resonator, reset time constant, and power filtering.
- Digital & I/O supply voltage operation from 2.5V to 3.6V.

- IOX-UART library is included to accelerate project inclusion using Arduino, platformio, or other development environments.
- Optional breakout development board with pogo pin fixture connectivity supports DIY custom programming and debugging using the low cost ST-LINK V2 debugger from STMicro.
- Other serial bus options available: 2-wire I2C and 4-wire SPI. See ABBY-IOX-I2C and ABBY-IOX-SPI versions.

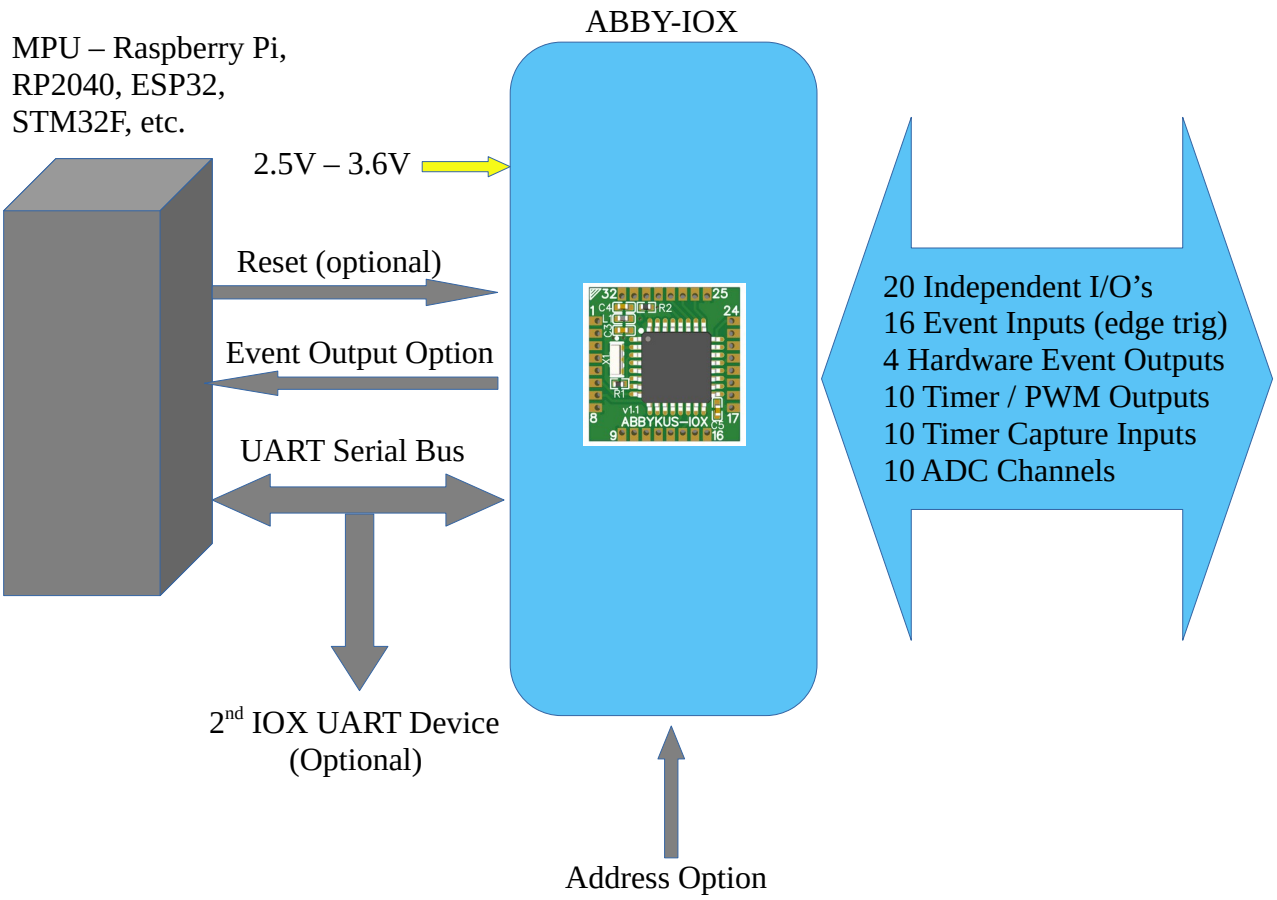


Figure 1. Basic Block Diagram

Table of Contents

INTRODUCTION

In many electronic project designs, even with MPU’s and MCU’s that sport a large number of general purpose I/O’s, it often happens that more resources are needed.

The ABBY-IOX I/O co-processor offers a low cost and flexible way to add the needed functionality without additional development time and expense.

The ABBY-IOX greatly expands I/O capability via a Universal Asynchronous Receiver Transmitter (UART) serial bus protocol into 20 parallel Input / Output bits. Each parallel bit can be programmed to be a digital input, output, or as an analog input for the embedded Analog to Digital converter.

Additionally inputs can be configured as event triggers with hardware and software event notification. Inputs can also provide waveform capture functions useful for operating encoders.

Outputs have various timer functions such as PWM with variable frequency & duty cycle, frequency and pulse width measurements, rotary encoder support, and much more.

UART Bus Basics

The ABBY-IOX-UART module version employs the familiar Universal Asynchronous Receiver-Transmitter (UART) 2-wire serial bus to connect to a system processor. Virtually all MPU's and MCU's support the UART serial bus standard. The IOX-UART operates in a half-duplex mode meaning that all communication on the bus are initiated by a master (system MPU / MCU). The IOX will only respond when the bus master begins a communication sequence. This mode allows for two IOX devices to be connected in parallel to the same UART bus and to a bus master.

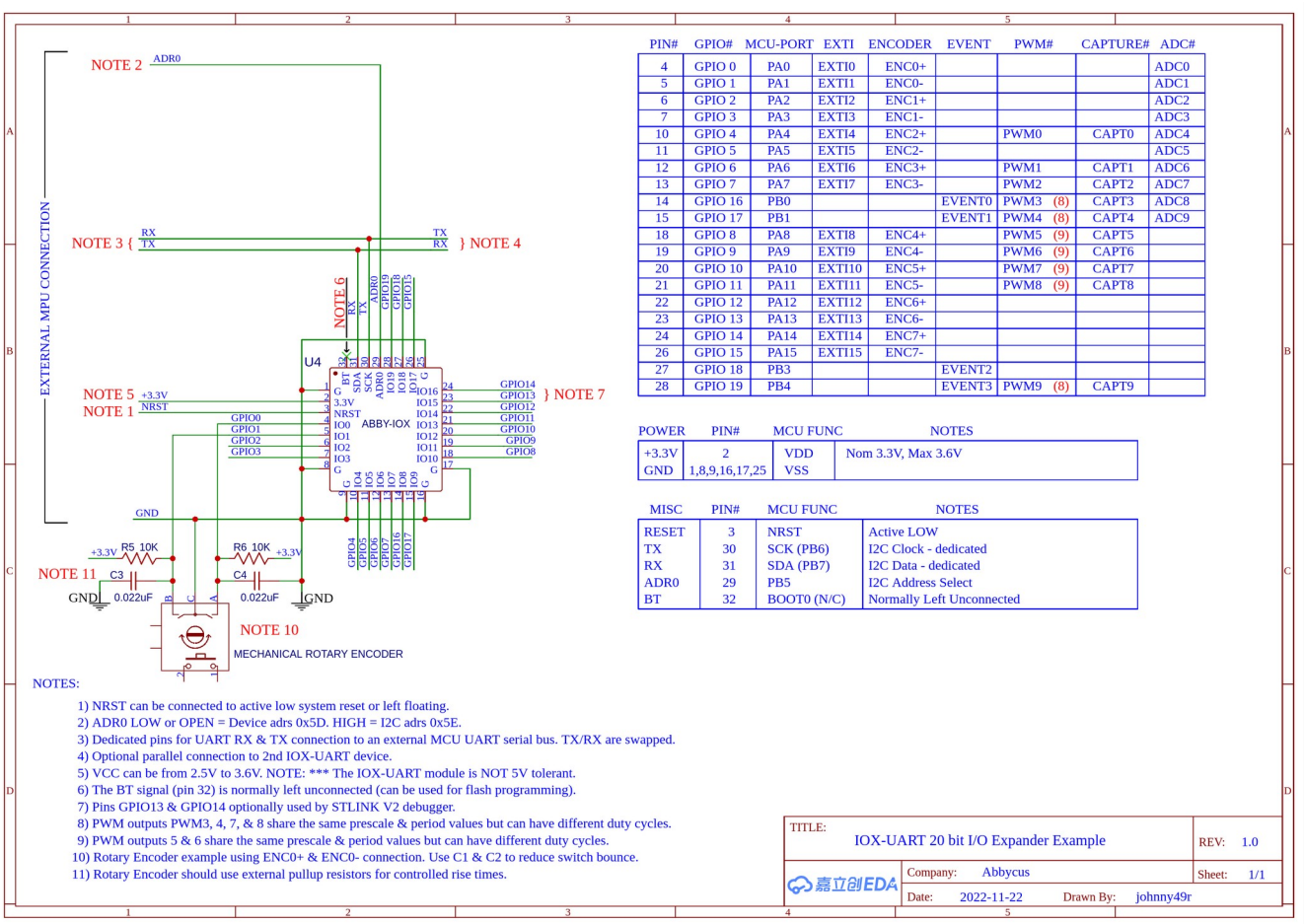
The UART bus consists of 2 wires called TX (transmit) and RX (receive). This simple 2-wire bus can support bit-rate speeds up to 6 MBaud. A more detailed description of the UART serial protocol can be found here.

NOTE: The default baud rate after power on or reset will be **115200** baud. Baud rate can be changed using the SET UART BAUDRATE function.

Each device on the bus responds to an address bit which is passed in the MSB of the command byte. The IOX-UART supports dual addressing using an address select input pin (**ADR0**). When the address bit in the MSB of the command byte matches the state of the ADR0 pin, communication can occur between master and an IOX slave device. Note that ADR0 pin can be left floating or connected to ground to use the default address (0).

Note: The TX signal from the master device should connect to the RX pin on the IOX-UART, and the RX signal from the master device should connect to the TX pin on the IOX-UART.

When connecting two IOX-UART devices together, connect TX on the 1st device to TX on the 2nd device, and connect RX of the 1st device to RX on the 2nd device (parallel signal connection). See also Figure 2 below.



PIN#	GPIO#	MCU-PORT	EXTI	ENCODER	EVENT	PWM#	CAPTURE#	ADC#
4	GPIO 0	PA0	EXTI0	ENC0+				ADC0
5	GPIO 1	PA1	EXTI1	ENC0-				ADC1
6	GPIO 2	PA2	EXTI2	ENC1+				ADC2
7	GPIO 3	PA3	EXTI3	ENC1-				ADC3
10	GPIO 4	PA4	EXTI4	ENC2+		PWM0	CAPT0	ADC4
11	GPIO 5	PA5	EXTI5	ENC2-				ADC5
12	GPIO 6	PA6	EXTI6	ENC3+		PWM1	CAPT1	ADC6
13	GPIO 7	PA7	EXTI7	ENC3-		PWM2	CAPT2	ADC7
14	GPIO 16	PB0			EVENT0	PWM3 (8)	CAPT3	ADC8
15	GPIO 17	PB1			EVENT1	PWM4 (8)	CAPT4	ADC9
18	GPIO 8	PA8	EXTI8	ENC4+		PWM5 (9)	CAPT5	
19	GPIO 9	PA9	EXTI9	ENC4-		PWM6 (9)	CAPT6	
20	GPIO 10	PA10	EXTI10	ENC5+		PWM7 (9)	CAPT7	
21	GPIO 11	PA11	EXTI11	ENC5-		PWM8 (9)	CAPT8	
22	GPIO 12	PA12	EXTI12	ENC6+				
23	GPIO 13	PA13	EXTI13	ENC6-				
24	GPIO 14	PA14	EXTI14	ENC7+				
26	GPIO 15	PA15	EXTI15	ENC7-				
27	GPIO 18	PB3			EVENT2			
28	GPIO 19	PB4			EVENT3	PWM9 (8)	CAPT9	

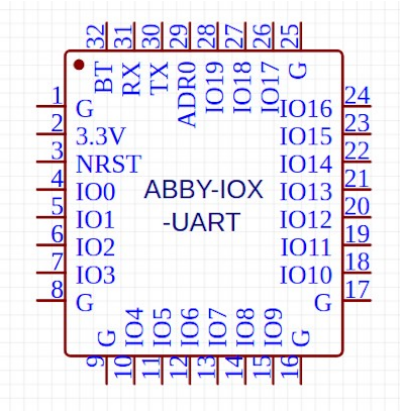
POWER	PIN#	MCU FUNC	NOTES
+3.3V	2	VDD	Nom 3.3V, Max 3.6V
GND	1,8,9,16,17,25	VSS	

MISC	PIN#	MCU FUNC	NOTES
RESET	3	NRST	Active LOW
TX	30	SCK (PB6)	I2C Clock - dedicated
RX	31	SDA (PB7)	I2C Data - dedicated
ADR0	29	PB5	I2C Address Select
BT	32	BOOT0 (N/C)	Normally Left Unconnected

Figure 2. UART Example Schematic and Pin Description.

IOX Package & Pinout

The IOX-UART package is a 1.6 cm square module with 32 pins numbered counterclockwise. Pin types are as follows:



S – Supply Pin

IO – Programmable Input / Output Pin

O – Output only pin.

I – Input only Pin

B – Dedicated Boot0 Pin

Pin #	Type	Pin Function(s)
1	S	Ground
2	S	Power supply voltage, 2.5V (minimum) to 3.6V (maximum)
3	I	Reset, active low, 500 ns minimum pulse width. Can be left open.
4	IO	GPIO 0 / ETXI0 / ADC0 / ENC0 +
5	IO	GPIO1 / EXTI1 / ADC1 / ENC0 -
6	IO	GPIO2 / EXTI2 / ADC2 / ENC1 +
7	IO	GPIO3 / EXTI3 / ADC3 / ENC1 -
8	S	Ground
9	S	Ground
10	IO	GPIO4 / EXTI4 / PWM0 / CAPT0 / ADC4 / ENC2 +
11	IO	GPIO5 / EXTI5 / ADC5 / ENC2 -
12	IO	GPIO6 / EXTI6 / PWM1 / CAPT1 / ADC6 / ENC3 +
13	IO	GPIO7 / EXTI7 / PWM2 / CAPT2 / ADC7 / ENC3 -
14	IO	GPIO16 / EVENT0 / PWM3 / CAPT3 / ADC8
15	IO	GPIO17 / EVENT1 / PWM4 / CAPT4 / ADC9
16	S	Ground
17	S	Ground
18	IO	GPIO8 / EXTI8 / PWM5 / CAPT5 / ENC4 +
19	IO	GPIO9 / EXTI9 / PWM6 / CAPT6 / ENC4 -
20	IO	GPIO10 / EXTI10 / PWM7 / CAPT7 / ENC5 +
21	IO	GPIO11 / EXTI11 / PWM8 / CAPT8 / ENC5 -
22	IO	GPIO12 / EXTI12 / ENC6 +
23	IO	GPIO13 / EXTI13 / ENC6 -
24	IO	GPIO14 / EXTI14 / ENC7 +
25	S	Ground
26	IO	GPIO15 / EXTI15 / ENC7 -
27	IO	GPIO18 / EVENT2
28	IO	GPIO19 / EVENT3 / PWM9 / CAPT9
29	I	ADR0 – Device address select pin. Can be left open.
30	O	TX – dedicated transmit output (tri-stated when transmitter is disabled)
31	I	RX – dedicated receiver input.

32	B	Boot0 programming pin - dedicated. Normally left open.

Command Description

Serial communication begins when the master sends a command byte. The lower 7 bits is the command (see table below), while the most significant bit is reserved as an address bit. The IOX device matches the address bit with the **ADR0** pin and if these two bits match, communication can continue. The master can then send any optional parameter bytes and the command will initiate some action from the IOX-UART device followed by one or more bytes returned from the IOX device.

Below are a list of command numbers and their meaning.

Command #	Description	Bytes Written	Bytes Read
1	WHO-AM-I	0	2
2	READ STATUS	1	3
3	CONFIGURE GPIO(s)	4	1
4	GET GPIO CONFIGURATION	1	2
5	WRITE GPIOS	2	1
6	TOGGLE GPIOS	2	1
7	READ GPIO	1	2
8	READ GPIO ALL	0	4
9	CONFIGURE EVENT OUTPUT	2	1
10	START PWM OUTPUT	5	1
11	UPDATE PWM DUTY CYCLE	2	1
12	CONFIGURE ADC INPUT(s)	2	1
13	START ADC CONVERSION	1	1
14	READ ADC CONVERSION	1	9
15	START CAPTURE	3	1
16	READ CAPTURE	1	5
17	CONFIGURE ENCODER	1	1
18	READ ENCODER	1	6
19	SLEEP	1	0
20	SET UART BAUDRATE	3	0
21	GET VERSION INFO	0	5

IOX-UART Library Description & Usage

All communication to the IOX-UART can be made using the supplied library consisting of two C++ files named ***iox_uart.h*** and ***iox_uart.cpp***.

Installation is simple – just copy these files in the project source folder and perform the following steps:

1. Include the library into your program with the statement ***#include "ioc_uart.h".***

2. Create a library object with the following statement: **IOX_UART iox_uart**. The optional address of the device depends on the state of the **ADR0** pin.

3. Perform a library call to initialize the IOX-UART as follows:

bool ret = **iox_uart.init(***uint8_t* ***uart_number****, uint8_t* ***PIN_TX****, uint8_t* ***PIN_RX****, uint32_t* ***BAUD-RATE)***

'uart_number' is the UART bus controller number. Many MCU's have more than one UART controller, for example the ESP32 has multiple controllers that are selected by number (typically 0 or 1).

'PIN_*TX*' and **'PIN_*RX*'** depend on the hardware pins specific to the MPU's UART bus. EXAMPLE: Using the ESP32 MPU, RX is pin 17 and TX is pin 18. Note that these pin assignments are somewhat arbitrary and depends on the master MPU and user assignments.

'BAUD_RATE' *is the UART baud rate of the master and* can be up to 6 Mbaud.

Examples

ret = iox_uart.init***(****1****,*** *18****,*** *17, 115200****)*** *// init UART 1 with baud-rate of 115200*

The ***init()*** function should return ***'true'*** if initialization is successful. If the result is false, check connections of the ***RX*** and ***TX*** pins. Also check that the ***uart_number*** is not being used already, and try the default baud-rate of 115200.

WHO_AM_I

bool **whoAmI(**uint8_t **iox_adrs)**

Description

The serial bus master can verify communication at any time using the WHO_AM_I function. The master communicates with the IOX-UART device and receives an arbitrary number (0x5D). The MSB

of the return value reflects the address of the device. Example: If ADR0 == 1 the returned value is 0xDD.

The library function simplifies this by returning a true or false Boolean value.

Parameters

‘iox_adrs’ is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

Returns

Returns ***true*** if communication is successful, or ***false*** otherwise.

READ STATUS

uint8_t **read_status(**uint8_t **iox_adrs**, SYS_STATUS * **sys_status,** uint8_t **field_mask)**

Description

A call to this function returns an 8-bit *STATUS* byte and a 16-bit *FIELD* word that are written into the **sys_status** structure.

Parameters

‘iox_adrs’ is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

‘sys_status’ is a pointer to a SYS_STATUS structure shown as follows:

```
typedef struct {                    // SYS_STATUS struct template
    uint8_t status;
    uint16_t field;
} SYS_STATUS;
SYS_STATUS sys_status;              // create a sys_status object
```

‘field_*mask*’ is a bit-wise byte that determines the status field returned. See the **FIELD WORD INTERPRETATION** table below for possible values. Also note that only one FIELD MASK value is allowed during each call to READ STATUS.

Examples

1) **err = read_status(0, &sys_status, FIELD_ERROR)** returns the current status and the ERROR field requested by the field_mask.

2) **err = read_status(0, &sys_status, FIELD_ENCODER)** returns the current status and any active ENCODERS in the FIELD word.

The function returns an ERROR CODE byte interpreted as follows:

0x00 **ERR_NONE** – no errors pending.

0x01 **ERR_PARAM** – parameter out of range occurred during the previous command.

FIELD MASK Macro Definitions:

0x01 **FIELD_ERROR** – returns system error field (0 if no errors).

0x02 **FIELD_EXTI** – returns a channel map of any externally triggered inputs. This action will clear the channel map and reset the optional hardware event output.

0x08 **FIELD_ADC_READY** – returns a channel map of ADC conversions that can be read.

0x10 **FIELD_CAPTURE** – returns a channel map of input captures that have occurred.

0x20 **FIELD_ENCODER** – returns a channel map of any encoders triggered by movement.

STATUS BYTE

7	
6	
5	1 = ENCODER movement detected.
4	1 = INPUT CAPTURE event has been triggered. This bit cleared by calling *READ_CAPTURE.*
3	1 = ADC CONVERSION complete. This bit is cleared after calling *READ_ADC.*
2	1 = ADC BUSY – Conversion in progress. This bit is automatically set & cleared.
1	1 = EXTI event has been triggered. This bit is cleared after calling *READ_EXTI.*
0	1 = ERROR - check error codes. This bit & *error bits are cleared after READ_STATUS.*

FIELD WORD INTERPRETATION

	Mask=0x01	**Mask=0x02**	**Mask=0x08**	**Mask=0x10**	**Mask=0x20**
BIT	**ERROR**	**EXTI**	**ADC CONV**	**INPUT CAPT**	**ENCODER**
15		EXTI15			
14		EXTI14			
13		EXTI13			
12		EXTI12			

11		EXTI11			
10		EXTI10			
9		EXTI9	ADC9	INCAP9	
8		EXTI8	ADC8	INCAP8	
7		EXTI7	ADC7	INCAP7	ENC7
6		EXTI6	ADC6	INCAP6	ENC6
5		EXTI5	ADC5	INCAP5	ENC5
4		EXTI4	ADC4	INCAP4	ENC4
3	1 = Command Timeout	EXTI3	ADC3	INCAP3	ENC3
2	1 = Serial Mode Failure	EXTI2	ADC2	INCAP2	ENC2
1	1 = GP Failure	EXTI1	ADC1	INCAP1	ENC1
0	1 = Parameter error – out of range or invalid value.	EXTI0	ADC0	INCAP0	ENC0

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

CONFIGURE GPIOS

void **config_gpios(**uint8_t **iox_adrs**, uint32_t **gpio_map,** uint8_t **gpio_config)**

Description

Configure one or more GPIO's with the same I/O attributes. Up to 20 GPIO's can be set as inputs or outputs using this single command.

On power up or after a reset, all un-dedicated GPIO's will default to analog input mode to minimize power consumption and avoid conflicts with attached signals.

Parameters

'iox_adrs' is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

'gpio_map' is a bit-wise map of all 20 GPIO's to configure as follows:

31 : 20	19 : 0
Unused	GPIO 19 : 0

Each bit, if set to 1, will configure the associated GPIO with the ***gpio_config*** value.

'gpio_config' is an encoded 6 bit value creating the type of I/O configuration requested.

The library header file *iox_uart.h* contains macro constants which can be O'rd together to create various I/O configurations as follows:

BASIC INPUT / OUTPUT Macro Definitions

IO_OUTPUT_PP – I/O is set as a push-pull output type.

IO_OUTPUT_OD – I/O set as an open-drain output type.

IO_INPUT – I/O is set as an input.

IO_LOW_POWER – I/O is set as an analog input type (low power ADC input).

OUTPUT SPEED ATTRIBUTE Macro Definitions

IO_SPEED_LOW – Lowest current drive output up to 2MHz.

IO_SPEED_MED – Medium speed drive output up to 10MHz.

IO_SPEED_HIGH – High speed drive output up to 50MHz.

INPUT ATTRIBUTE Macro Definitions

NOTE: Only GPIO0 – GPIO15 can be programmed as external interrupts. GPIO16 – GPIO19 will ignore these attributes.

IO_EXTI_DISABLE – Input events disabled.

IO_EXTI_RISING – Event on input rising edge.

IO_EXTI_FALLING – Event on input falling edge.

IO_EXTI_BOTH – Event on either edge.

INPUT & OUTPUT ATTRIBUTE Macro Definitions

IO_NO_PUPD – No pull-up or pull-down resistors used.

IO_PULLUP – Weak pull-up resistor to VCC (approx 40K ohms).

IO_PULLDOWN – Weak pull-down resistor to ground (approx 40K ohms).

NOTE: Weak pull-up and pull-down resistors (approx 40K ohms) can be added to either input or output configurations. Note that It's not possible to set both pull-up and pull-down resistors together.

Examples

1. ***config_gpios(0, 0xFF, IO_OUTPUT_OD | IO_PULLUP)*** creates an open drain output with a weak pull-up resistor on GPIO's 0-7.

2. ***config_gpios(0, 0xFFFF8, IO_INPUT | IO_PULLUP | IO_EXTI_RISING)*** *creates an input with a weak pull-up resistor and a rising edge event trigger for the GPIO range of 3 – 19.*

3. ***config_gpios(0, GPIO1 | GPIO4, IO_LOW_POWER)*** *creates an analog input for GPIO1 and GPIO4.*

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

GET GPIO CONFIGURATION

uint8_t **get_gpio_config(**uint8_t **iox_adrs**, uint8_t **gpio_num)**

Description

Returns the current configuration of the specified GPIO number.

Parameters

'iox_adrs' is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

'gpio_num' is the logical GPIO number with a value of 0 – 19.

Returns

The returned configuration is encoded into a 6 bit value which is a mirror of the I/O configuration value sent by the **CONFIGURE GPIOS** command (see above).

7 : 6	5	4 : 0
n/a	Mode	Attribute(s)

NOTE: If an error occurs the function will return a value of 0xFF (255 decimal).

WRITE GPIOS

uint8_t **write_gpios(**uint8_t **iox_adrs**, uint32_t **gpio_map,** uint32_t **state_map)**

Description

This command writes one or more GPIO's with the same '0' or '1' value.

Parameters

'iox_adrs' is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

'gpio_map' is a bit-wise map of all GPIO's. A '1' bit associated with the logical GPIO will select that GPIO for writing (lower 20 bits).

'state_map' is a bit-wise map of the state to write for each GPIO position (lower 20 bits).

Examples

uint32_t gpio_map = 0xFFFFF; // write all 20 GPIO's

uint32_t state_map = 0x55555; // write even bits with '1', odd bits with '0'

ret = write_gpios(0, gpio_map, state_map); // write gpio's

---

uint32_t gpio_map = 0b10001000000000100001; // select GPIO's 0, 5, 15, & 19

uint32_t state_map = 0xFFFFF; // write selected GPIO's with a '1'

ret = write_gpios(0, gpio_map, state_map); // write gpio's

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

TOGGLE GPIO(s)

uint8_t **toggle_gpios(**uint8_t **iox_adrs**, uint32 **gpio_map)**

Description

One or more I/O outputs can be toggled (inverted) using this single command.

Parameters

‘iox_adrs’ is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

‘gpio_map’ is a bitwise map of all GPIO’s (lower 20 bits). A ‘1’ in the associated GPIO position will cause that output to be toggled.

31 : 20	19 : 0
Unused	GPIO’s 19 : 0

Examples

uint32_t toggle_map = 0x00105;

ret = toggle_gpios(0, toggle_map) // invert GPIO outputs 0, 2, and 8

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

READ GPIO

uint8_t **read_gpio(**uint8_t **iox_adrs**, uint8_t **gpio_num,** uint8_t * **gpio_value)**

Description

Any GPIO pin can be read using this command. If the GPIO is configured as an output, the function will return the current value of the output pin.

Parameters

‘iox_adrs’ is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

‘gpio_num’ is the logical GPIO to be read and must be in the range of 0 – 19, otherwise the function will set the ERROR bit in the STATUS register and a ERR_PARAM bit in the ERROR REGISTER (see **READ STATUS** and **ERROR HANDLING**).

‘gpio_value’ is a pointer to an 8 bit memory location where the state of the GPIO pin will be written.

Examples

```
uint8_t gpio_val;

ret = read_gpio(0, 13, (uint8_t *) &gpio_val);    // gpio_val will have the state of GPIO 13
```

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

READ GPIO ALL

uint8_t **read_gpio_all(**uint8_t **iox_adrs**, uint32_t * **gpio_map)**

Description

All I/O’s configured as inputs can be read with this command. GPIO’s configured as output will return the state of the output.

Parameters

‘iox_adrs’ is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

‘gpio_map’ is a pointer to a 32 bit memory location where the 20 bit map will be written.

31 : 20	19 : 0
Unused	GPIO 19 : GPIO 0

If the any GPIO is configured as an output, the function will return the current value of the output pin.

Examples

uint32_t gpio_map;

ret = read_gpio_all(0, (uint32_t *) &gpio_map); // map written into gpio_map memory loc

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

CONFIGURE EVENT OUTPUT

uint8_t **config_event_output(**uint8_t **iox_adrs**, uint8_t **event_type,** uint8_t **event_num)**

Description

This command configures one of four possible event outputs (EVO0 - EVO3). When an event occurs the designated GPIO output will go high. When **READ STATUS** is called, the hardware event output will be cleared.

Parameters

'iox_adrs' is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

'event_type' parameter is one of the following:

0 - **EVENT_NONE** clears the specified event output.

1 – **EVENT_EXTI** sets the event output when a change occurs on one or more GPIO's configured as inputs triggered by edge changes (see CONFIGURE GPIOS).

2 – **EVENT_ADC** sets the specified event output when an ADC conversion is complete and is ready to be read (see READ ADC).

3 – **EVENT_CAPTURE** sets the specified event output when an INPUT CAPTURE operation is complete and is ready to be read (see READ CAPTURE).

4 – **EVENT_ENCODER** sets the specified event output when a ROTARY ENCODER movement is detected and is ready to be read (see READ ENCODER).

When an event occurs, the event output can be cleared by calling **READ STATUS** with the 'field_mask' set to the appropriate value.

'event_num' is the logical event output pin and should be in the range 0 – 3.

Example

ret = config_event_output(0, EVENT_ADC, 0); // event 0 output when ADC conversion is done.

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

START PWM OUTPUT

uint8_t **start_PWM(**uint8_t **iox_adrs**, uint8_t **pwm_num,** uint16_t **clk_div,** uint16_t **pwm_freq,** uint16_t **pwm_duty,** bool **polarity**

Description

The command configures a GPIO pin to output a desired Pulse Width Modulated (PWM) waveform.

Parameters

‘iox_adrs’ is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

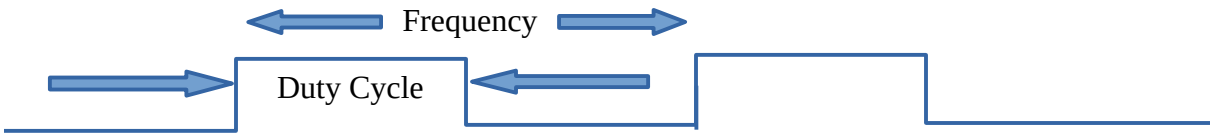
‘pwm_num’ *is t*he logical PWM pin (0 - 9) to output the desired PWM waveform.

‘clk_div’ is the PWM timer prescaler value. The timer runs from a base clock of 48 MHz and is first divided by this value. The division factor is ‘n+1’ so a value of 0 means that the base clock will be divided by 1 so the PWM timer clocks at 48 MHz. A value of 47 means that the base clock will be divided by 48 so the base clock is 1 MHz, etc.

‘pwm_freq’ is the frequency (or period) of the PWM waveform in prescaled clock cycles +1. So a value of 999 would yield a period of 1000 clocks.

‘pwm_duty’ is the number of clock cycles that define the ‘on-time’ of the PWM waveform.

‘polarity’ inverts the waveform output if true.



Examples

// Start PWM on IOX device 0, PWM2 channel, clk = 1 us, frequency = 2000 clocks,

// 1000 us duty cycle (50%), non-inverted

ret = start_PWM(0, 2, 47, 1999, 1000, false);

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

See also **NOTES ON PWM GENERATION**.

UPDATE PWM

uint8_t **update_PWM(**uint8_t **iox_adrs**, uint8_t **pwm_num,** uint16_t **pwm_duty)**

Description

The duty cycle of a PWM waveform can be updated synchronously to achieve variable pulse width (servo control). The change to the duty cycle occurs at the beginning of the next PWM period.

Note that the PWM channel should have been initialized by calling START PWM OUTPUT function.

Parameters

'iox_adrs' is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

'pwm_num' is the logical PWM channel and can be in the range of 0 – 9.

'pwm_duty' has a range of 0 – 65535 which determines the on-time during the waveform period.

Examples

ret = update_PWM(0, PWM2, 666); // change duty cycle on PWM2 to 666 usec

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

CONFIGURE ADC CHANNEL

bool **config_ADC(**uint8_t **iox_adrs**, uint16_t **adc_chnls,** uint8_t **adc_resol)**

Description

The IOX-UART device contains one 12-bit configurable Analog to Digital Converter (ADC) which can convert analog voltages from 0 to 3.3V to a digital value.

The function configures a group of GPIO pins as inputs to the ADC. Up to 10 channels can be multiplexed into the ADC.

Parameters

‘iox_adrs’ is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

‘adc_chnls’ is a 16 bit map where the low order 10 bits represent all logical ADC channels. A ‘1’ bit will configure the logical ADC channel as a high impedance analog pin. Multiple channels will be converted starting from the lowest channel number up to the highest.

‘adc_resol’ is the converter’s width in bits. Valid values are 6, 8, 10, and 12. Higher values yield finer resolution but have longer conversion times. The ADC is a successive approximation type meaning that the conversion time varies over a span of about 12:1. Therefore the conversion time for a 12 bit width is in the range of 1 microsecond to 12 microseconds.

The reference voltage for all channels is the VCC of the device (typically 3.3V). Accuracy of measurement depends on the accuracy of the external voltage applied to the IOX power input pin 2.

Examples

```
ret = config_ADC(0, 1, ADC_RESOL_12);   // config ADC chnl 1 for 12 bit conversion.
```

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

START ADC CONVERSION

uint8_t **start_ADC(**uint8_t **iox_adrs**, uint16_t **adc_chnls,** uint16_t **num_samples)**

Description

This function starts an ADC conversion sequence on the designated channels. Configured ADC channels are scanned from the lowest numbered to the highest.

Conversion values for each channel can be averaged over many samples to achieve more accurate and stable results (oversampling).

Parameters

‘iox_adrs’ is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

‘adc_chnls’ is a 16 bit map where the low order 10 bits represent all possible ADC inputs. A ‘1’ bit in the map will start the A/D conversion on the associated logical channel. Each conversion channel should be previously configured by the **CONFIGURE ADC CHANNEL** function.

‘num_samples’ is a 16 bit value allowing up to 65535 samples for each channel. The averaged result is obtained using the **READ ADC** function. The number of samples should be a minimum of 1. A value of 0 will result in an error status with a parameter error code.

A conversion sequence begins by setting STATUS_ADC_BUSY in the STATUS byte indicating the ADC has been started but conversion is not yet complete.

When all conversions in a sequence have completed, the STATUS_ADC_BUSY will be cleared and the STATUS_ADC_READY will be set. See **READ STATUS** command.

Examples

```
uint16_t adc_map = 0b1000001001;     // read channels 0, 3, and 9

ret = start_ADC(0, ADC0, adc_map, 256);  // start conversion and average over 256 samples.
```

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

READ ADC

uint8_t **read_ADC(**uint8_t **iox_adrs**, uint8_t **adc_chnl,** ADC_CONVERT * **adc_conv)**

Description

Following a START ADC command the ADC will begin reading conversions and averaging samples. When all conversions have been completed the STATUS_ADC_BUSY will be cleared and the STATUS_ADC_READY will be set.

Parameters

‘iox_adrs’ is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

‘adc_chnl’ specifies the logical channel number 0 – 9.

‘adc_conv’ is a pointer to an ADC_CONVERT structure (included in iox_uart.h) as follows:

```
typedef struct {              // ADC_CONVERT struct template
    uint16_t mean;            // mean (averaged) result of the conversion(s)
    uint16_t min;             // minimum value read.
    uint16_t max;             // maximum value read.
    uint16_t samples;         // number of samples converted.
} ADC_CONVERT;
ADC_CONVERT adc_conv;         // create a adc_conv object
```

When all channels involved in the conversion have been read, the STATUS_ADC_READY bit in the STATUS byte will be cleared and another conversion process can begin.

Examples

ADC_CONVERT adc_conv;

ret = read_ADC(0, ADC9, &adc_conv); // read conversion results of ADC9 input

Serial.printf(“ADC9 average= %d\n”, adc_conv.mean);

Notes:

Conversion from ADC reading to volts is as follows:

ADC bit resolution:

12 bit: 2^12 = 4096

10 bit: 2^10 = 1024

8 bit: 2^8 = 256

6 bit: 2^6 = 64

Volts = (VCC (3.3V) / bit resol) * reading.

Example: (3.3 / 4096) * 1241 = 0.9998 V

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

START CAPTURE

uint8_t **start_capture(**uint8_t **iox_adrs**, uint8_t **capt_chnl,** uint8_t **trig_edge,** uint8_t **capt_type)**

Description

The START CAPTURE command configures the specified channel for Input Capture and arms the capture trigger to begin measuring on the next specified edge. An internal timer measures the time between two consecutive edges (frequency) or between two opposite edges (pulse width).

The resulting measurement is saved as the number of timer clocks between edges. The timer clock is 48 MHz or 20.8333 nanoseconds per tick.

Parameters

'iox_adrs' is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

'capt_chnl' specifies the channel to be used (0 – 9).

'trig_edge' defines the edge to be used to begin the measurement. The trig_edge macros are defined in the *ioxi2c.h* file and can be one of the following:

0 – **INPUT_CAPT_RISING**: measurement starts at the next rising edge of the signal.

1 – **INPUT_CAPT_FALLING**: measurement starts at the next falling edge of the signal.

5 – **INPUT_CAPT_BOTH**: measurement starts at the next edge change (rising or falling).

'capt_type' determines the measurement type:

0 – CAPTURE_TYPE_PW: measures time between opposite edges (pulse width):

1 – CAPTURE_TYPE_FREQ: measures time between same edges (frequency):

Examples

// start a pulse width measurement on CAPT1 starting on the next rising edge

ret = start_capture(0, CAPT1, INPUT_CAPT_RISING, CAPTURE_TYPE_PW);

When the capture measurement has completed, the FIELD_CAPTURE bit in the STATUS byte will be set. See **READ STATUS**.

Additionally, if the EVENT OUTPUT has been configured for CAPTURE, the designated GPIO will be set high. See **CONFIGURE EVENT OUTPUT**.

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

See also **NOTES ON INPUT CAPTURE MEASUREMENTS**.

READ CAPTURE

uint32_t **read_capture(**uint8_t **iox_adrs**, uint8_t **capt_chnl)**

Description

This command returns the result of the last START CAPTURE command. The returned 32 bit value represents the number of clock periods measured during the capture operation.

The capture timer clock runs at 48 MHz so each clock period is equivalent to 20.8333 nanoseconds or 0.0208333 microseconds.

A READ CAPTURE is typically called when the STATUS byte indicates that a capture measurement has been completed. See **READ STATUS**.

Parameters

‘iox_adrs’ is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

‘capt_chnl’ specifies the logical capture channel to read (0 – 9).

Examples

capt = read_capture(0, CAPT1); // read last measurement on CAPT1 channel

As an example, if the capt count == 99, the time would be 2.0625 microseconds (99 * 0.028333).

Returns

The returned value will be 0 if an error occurs.

CONFIGURE ENCODER

uint8_t **config_encoder(**uint8_t **iox_adrs**, uint8_t **enc_chnl)**

Description

This command assigns and configures an adjacent pair of GPIO’s for use as a Rotary Encoder input.

The IOX device can support up to 8 rotary encoders that provide direction, rotational position, and rotational velocity when rotated. Rotary encoders generate 2 signals that are 90 degrees apart (Clk / Data, +/-, A / B, etc.) and require 2 inputs to the IOX device.

Some encoders use optical sensors whereas lower cost encoders may be mechanical switch encoders (KY-040 type for example). See Figure 2 for an example using a mechanical encoder type.

Parameters

‘iox_adrs’ is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

‘enc_chnl’ defines the GPIO pair that is connected to the encoder’s +/- pair. See **IOX Package & Pinout**. Also see **Figure 2** for an example of encoder connection and note the external pull-up resistors and capacitors used to debounce noisy signals.

Whenever the encoder is rotated the **FIELD_ENCODER** bit in the **STATUS** byte will indicate encoder motion. Optionally a designated hardware output pin can notify encoder motion (see **CONFIGURE EVENT OUTPUT**).

Examples

Ret = config_encoder(0, ENC1); // attach an encoder to GPIO’s 2 & 3.

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

READ ENCODER

uint8_t **read_encoder(**uint8_t **iox_adrs**, uint8_t **enc_chnl,** ROT_ENC * **rot_enc)**

Description

Whenever an encoder is rotated, the **STATUS_ENCODER** bit in the **STATUS** byte will indicate encoder motion. A call to **READ STATUS** with the FIELD_ENCODER bit set in the field mask will return a map of which attached encoder was rotated.

Optionally a designated hardware output pin can notify encoder motion (see **CONFIGURE EVENT OUTPUT**).

Parameters

'iox_adrs' is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

'enc_chnl' is the logical channel to read (0 – 7).

'rot_enc' is a pointer to a ROT_ENC structure (defined in ioxi2c.h) as follows:

```
typedef struct {
    bool dir;           // true indicates CW rotation
    int16_t count;      // running count (reset to 0 when encoder is read)
    uint16_t speed;     // approx rotational speed in RPM
} ROT_ENC;
ROT_ENC rot_enc;  // create a rot_enc struct object
```

'dir' value is true if the encoder is rotated clockwise, false otherwise. NOTE: If this is opposite of the desired indication, the encoder signal pair can simply be reversed.

'count' value will increment or decrement depending on the direction of rotation. This value is reset to 0 after reading the encoder value.

'speed' value is an approximation of the rotational velocity expressed as RPM. NOTE: This may not be accurate or reliable when the encoder is reversed direction or rotation is unstable.

Examples

ROT_ENC rot_enc;

ret = read_encoder(0, ENC1, &rot_enc); // read last encoder values for ENC1

Serial.printf("count=%d\n", rot_enc.count); // print current count value

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

SLEEP

uint8_t **sleep(**uint8_t **iox_adrs**, uint8_t **wake_gpio)**

Description

The IOX module can save power by putting the device into sleep mode. Power consumption will be reduced to approximately 4ma in sleep mode. When the device is put into sleep mode the internal CPU clock will be stopped but I/O's and most peripheral controllers remain active.

Parameters

'iox_adrs' is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

'wake_gpio' specifies the GPIO that will wake the sleeping device.

Wake up from sleep can happen three different ways:

1) If NRST (pin 3) is pulled to ground and released, the device will re-boot and operate normally. This may not be desirable since the device will need to be reconfigured to the state it was in prior to sleep.

2) Wake up can be programmed to occur on activity from one of the GPIO pins specified by **'wake_gpio'**. This means that the device will remain in its current configuration and will not incur boot up time or re-configuration through the serial bus.

3) Wake up from UART bus activity. This functions the same way as option #2 except that the **'wake_gpio'** should be set to 255 (0xFF). This may not work if there are two devices on the UART bus since both devices would wake together.

Prior to issuing the **sleep** command the wake method **'wake_gpio'** should be configured to generate an interrupt by calling the **CONFIGURE GPIOS** command.

Examples

config_gpios**(**0**,** 0x8**,** IO_INPUT | IO_PULLUP | IO_EXTI_FALLING**);** // trig gpio 3 on falling edge

ret = sleep(0, 3); // sleep & wake on falling edge of GPIO3

Further power savings can be implemented by setting all unused GPIO pins to IO_LOW_POWER (Analog) inputs. See **CONFIGURE GPIOS**.

Returns

The sleep function does not return any data.

SET UART BAUDRATE (IOX-UART version only)

uint8_t **set_uart_baud(**uint8_t **iox_adrs,** uint32_t **baudrate)**

Description

The UART baudrate can be set at any time to match the desired baud rate of the master device.

The default baudrate after device reset is **115200** baud. After initial communication has been established at the default baudrate the master can send a SET UART BAUDRATE command to operate the IOX device at another speed.

NOTE: The IOX device will revert to its default baudrate following a power on or reset. If communication is lost, the master should try the default speed of 115200 baud to re-establish communication.

Parameters

‘iox_adrs’ is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

‘baudrate’ is the requested speed in bits/second.

Examples

uint8_t set_uart_baud(0, 250000); // IOX device 0, set new baudrate = 250000 baud

Returns

The SET UART BAUDRATE command normally will not return anything and the iox device will resume communication at the new baudrate.

GET VERSION INFO

uint8_t **get_version(**uint8_t **iox_adrs,** VERSION_INFO * **version)**

Description

Returns 4 bytes of information used to identify the serial bus type and firmware version number.

Parameters

‘iox_adrs’ is used to select an IOX device in a multi device configuration. Value should be 0 or 1.

‘version’ is a pointer to a VERSION_INFO structure (defined in iox_uart.h) as follows:

```
typedef struct {
    uint8_t reserved;       // for future use...
    uint8_t bus_version;    // 1 == I2C, 2 == UART, 3 == SPI
    uint8_t major_rev;      // major part of version number
    uint8_t minor_rev;      // minor part of version number
} VERSION_INFO;
VERSION_INFO version;   // version object
```

Examples

ret = iox_uart.get_version(0, &version);

Serial.printf(“Bus version=%d, version=%.02d.%.02d\n”, version.bus_version, version.major_rev, version.minor_rev);

Returns

The function will return a ERR_NONE (0x0) on successful completion or an error code (see **ERROR HANDLING**).

ERROR HANDLING

Errors can occur as a result of serial bus communication problems or as a result of command execution after the IOX receives a command.

Most functions return an 8-bit error code indicating if communication to the IOX device was successful. The following is a list of possible codes:

Hex Code	Description
0x00	ERR_NONE - Function successful
0x01	ERR_PARAM - Parameter out of range.
0x02	ERR_GP_FAILURE - General Failure
0x04	ERR_SERIAL_MODE - Serial bus mode error.
0x08	ERR_TIMEOUT - Communication timeout.

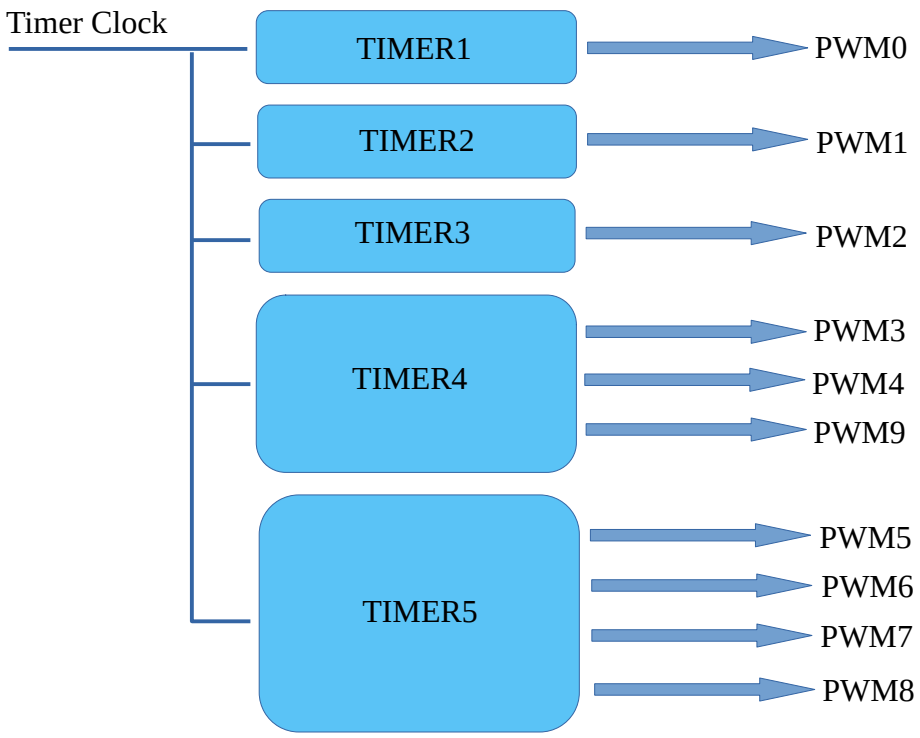
When an error occurs during serial bus communication, the library will return a bus failure error. The error is as follows:

When an error occurs during command execution (after receiving the command correctly) the IOX will assert STATUS_ERROR bit in the STATUS byte. Details about the error can be discovered by issuing a READ STATUS with the ‘field_mask’ set to STATUS_ERROR. See **READ STATUS**.

NOTES ON PWM GENERATION

PWM waveforms are created using timers embedded in the IOX MCU. There are a total of five timers that can be used for PWM generation. Three of these timers are independent from the others while the remaining two timers have multiple shared output channels.

Timers that have multiple outputs can have different duty cycle values but share the same pre-scaler and period frequency values. Common period frequency may not be an issue for servo control since multiple servos would likely require the same period frequency.



NOTES ON INPUT CAPTURE MEASUREMENTS

Input Capture measurements use timers embedded in the IOX MCU. There are a total of five timers that can be used for Input Capture. Three of these timers are independent from the others while the remaining two timers have multiple shared input channels.

NOTE: Timers with multiple capture inputs cannot be used simultaneously for multiple capture measurements or for PWM generation.

The clock for all timers used for Input Capture is 48 MHz. Capture measurements are reported as the number of clocks between two edge transitions. The measured time reported is an unsigned 32 bit value which limits the maximum measurement time to 89.48 seconds or 0.011 Hz.