ABBYCUS IOX  ABBY-IOX-I2C

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 I2C 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 I/O 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 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 a programmable output pin (hardware notification).
- 12 bit Analog To Digital Converter (ADC) with up to 10 multiplexed inputs. Range = 0 - 3.3 VDC typical. Conversions can be started via hardware or software.
- Support for 8 rotary encoders with direction, count, and rotational speed calculated.
- 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-I2C 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 UART and 4-wire SPI. See ABBY-IOX-UART and ABBY-IOX-SPI versions.

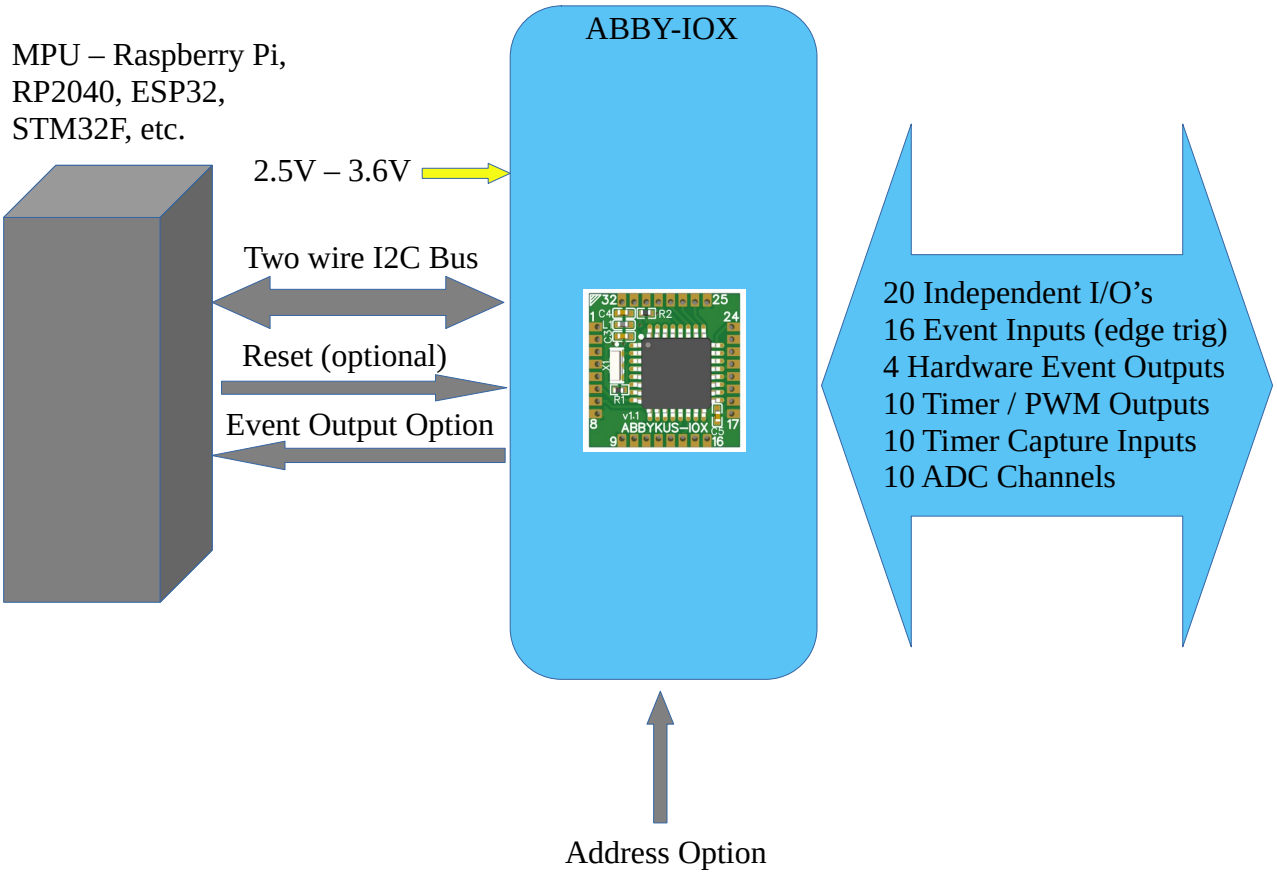


Figure 1. Basic Block Diagram

Table of Contents

Features....1
Figure 1. Basic Block Diagram....2
INTRODUCTION....6
I2C Bus Basics....7
Figure 2. I2C Example Schematic and Pin Description....7
IOX Package & Pinout....8
The IOX-I2C package is a 1.6 cm square module with 32 pins numbered counterclockwise. Pin types are as follows:....8
S – Supply Pin....8
IO – Programmable Input / Output Pin....8
I – Input only Pin....8
O – Output only pin....8
B – Dedicated Boot0 Pin....8
Command Description....10
IOX-I2C Library Description & Usage....11
Examples....11
Returns....11
WHO_AM_I....11
Description....12
Parameters....12
Returns....12
READ STATUS....12
Description....12
Parameters....12
CONFIGURE GPIO(s)....14
Description....14
Parameters....14
Examples....16
Returns....16
GET GPIO CONFIGURATION....16
Description....16
WRITE GPIOS....16
Description....17
Parameters....17
Examples....17
Returns....17
TOGGLE GPIO(s)....17
Description....17
Parameters....17
Examples....18
Returns....18
READ GPIO....18
Description....18
Parameters....18

Examples....19
Returns....19
READ GPIO ALL....19
Description....19
Parameters....19
Examples....19
Returns....20
CONFIGURE EVENT OUTPUT....20
Description....20
Parameters....20
Example....20
Returns....21
START PWM OUTPUT....21
Description....21
Parameters....21
Examples....21
Returns....22
UPDATE PWM....22
Description....22
Parameters....22
Examples....22
Returns....22
CONFIGURE ADC CHANNEL....22
Description....23
Parameters....23
Examples....23
Returns....23
START ADC CONVERSION....23
Description....23
Parameters....24
Examples....24
Returns....24
READ ADC....24
Description....24
Parameters....24
Examples....25
Returns....25
START CAPTURE....25
Description....25
Parameters....26
Examples....26
Returns....26
READ CAPTURE....27
Description....27
Parameters....27
Examples....27
Returns....27

CONFIGURE ENCODER....27
Description....27
Parameters....28
Examples....28
Returns....28
READ ENCODER....28
Description....28
Parameters....28
Examples....29
Returns....29
SLEEP....29
Description....29
Parameters....29
Examples....30
Returns....30
ERROR HANDLING....30
NOTES ON PWM GENERATION....31
NOTES ON INPUT CAPTURE MEASUREMENTS....32

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-I2C greatly expands I/O capability through a generic serial bus (I2C) into 20 parallel Input / Output pins. Each pin 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.

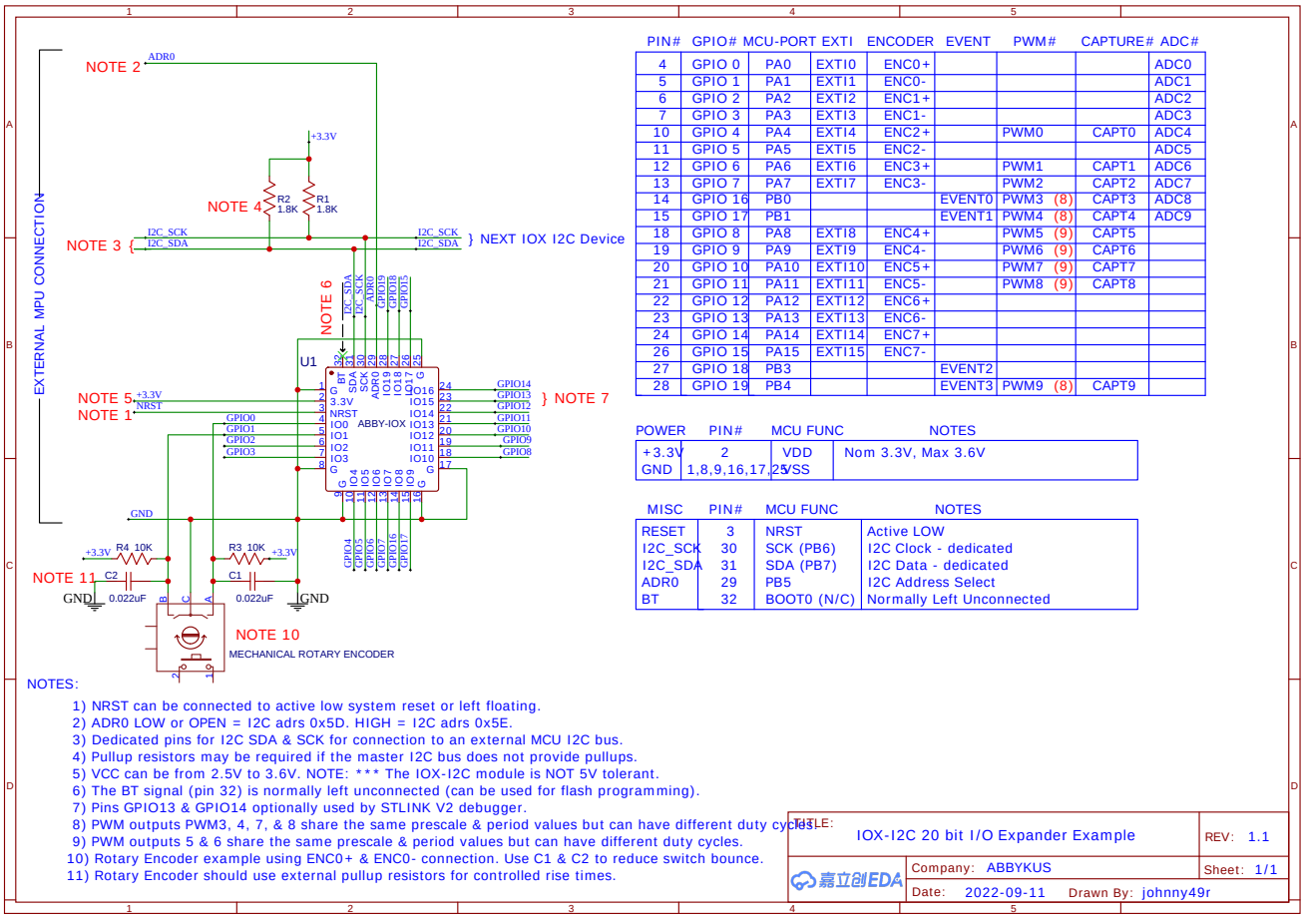
Outputs have various timer functions such as PWM with variable frequency & duty cycle, frequency and pulse width measurements, rotary encoder support, and much more.

I2C Bus Basics

The ABBY-IOX-I2C module version employs the familiar Inter-Integrated-Circuit (I2C) 2-wire serial bus used to connect to a system processor. Virtually all MPU's and MCU's support the I2C bus standard. The IOX-I2C acts as a bus slave device meaning that all communication on the bus are initiated by a master (system MPU / MCU).

The I2C bus consists of 2 wires called SCK (bus clock) and SDA (bus data). This simple 2-wire bus can support bit-rate speeds of 100KHz (standard mode), 400KHz (high speed mode), or up to 1MHz (fast mode plus). A more detailed description of the I2C bus can be found here.

The I2C bus is a party-line bus that can support multiple devices. Each device on the bus responds to a unique 7-bit address. The IOX-I2C supports a dual addressing scheme using an address select input pin (ADR0). The default bus address is 0x5D but the user can change this to 0x5E by connecting ADR0 (pin 31) to 3.3V. Note that ADR0 can be left unconnected to use the default address.



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
I2C_SCK	30	SCK (PB6)	I2C Clock - dedicated
I2C_SDA	31	SDA (PB7)	I2C Data - dedicated
ADR0	29	PB5	I2C Address Select
BT	32	BOOT0 (N/C)	Normally Left Unconnected

Figure 2. I2C Example Schematic and Pin Description.

IOX Package & Pinout

The IOX-I2C package is a 1.6 cm square module with 32 pins numbered counterclockwise. Pin types are as follows:

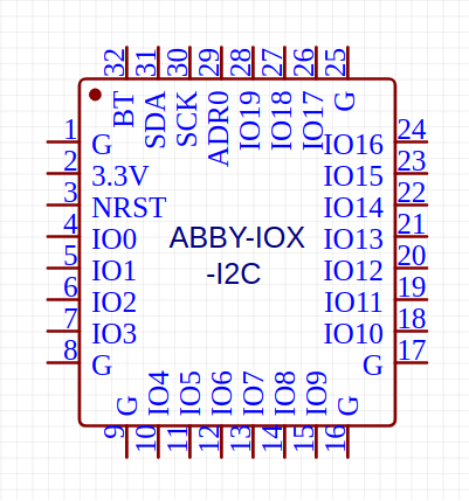
S – Supply Pin

IO – Programmable Input / Output Pin

I – Input only Pin

O – Output only pin.

B – Dedicated Boot0 Pin



Pin #	Type	Description
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 – I2C address select pin. Can be left open.
30	IO	I2C SCK – dedicated I2C signal.
31	IO	I2C SDA – dedicated I2C signal.
32	B	Boot0 dedicated programming pin. Normally left open.

Command Description

The ABBY-IOX-I2C uses a register paradigm to write and read data from the module where a register number is synonymous with a command. The I2C bus begins communication with the master sending the 7-bit slave address. The IOX slave device that recognizes the address will acknowledge an address match and communication can continue. The master must then send a second byte which will be a register (command) number to initiate some action from the IOX-I2C device. Optionally, depending on the register number, more data can be exchanged between master and slave. Below is a list of register numbers and their meaning.

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

IOX-I2C Library Description & Usage

All communication to the IOX-I2C can be made using a supplied library consisting of two C++ files named ***iox_i2c.h*** and ***iox_i2c.cpp***.

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

1. Include the library into your program with the statement ***#include "ioc_i2c.h".***

2. Create a library object with the following statement: **IOXI2C iox_i2c()**.

3. Perform a library call to initialize the I2C communication as follows:

bool ***ret = iox_i2c.init(****uint8_t* ***PIN_I2C_SDA,*** *uint8_t* ***PIN_I2C_SCK,*** *uint32_t* ***BUS_SPEED, &Wire)***

The initialization function will return 'true' if the call to the I2C bus controller is successful.

The pins ***SDA*** and ***SCK*** depend on the hardware pins specific to the system I2C bus. EXAMPLE: Using the ESP32 MPU, SCK is wired to pin 17 and SDA is wired to pin 18. Note that these pin assignments are somewhat arbitrary and depends on the user assignments.

The I2C ***BUS_SPEED*** can be 100000 (std speed), 400000 (fast speed), or up to 1000000 (fast plus speed). The IOX device has been tested up to 1 MHz bit rate.

The I2C bus controller designated by the name ***Wire*** is the default I2C bus controller for the standard Wire library used with Arduino, platformio, etc.

Examples

ret = iox_uart.init***(****1****,*** *18****,*** *17, 230400****)*** *// init UART 1 with baud-rate of 230400*

Returns

The function ***init()*** should return '*true*' indicating successful initialization. If the result is false, check connections of the ***SDA*** and ***SCK*** pins. Also check that ***Wire*** is not being used already, and try the standard bus speed of 100000.

WHO_AM_I

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

Description

The I2C bus master can verify communication at any time using the WHO_AM_I function. The master communicates with the IOX-I2C slave device and receives it's I2C bus address (0x5D or 0x5E).

The library function simplifies this by returning ***true*** if communication is successful, or ***false*** otherwise.

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 writes an 8-bit *STATUS* byte and a 16-bit *FIELD* word into a **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.

EXAMPLES:

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

2) **err = read_status(0, &sys_status, STATUS_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		EXTI3	ADC3	INCAP3	ENC3
2		EXTI2	ADC2	INCAP2	ENC2
1	1 = I2C bus error (timeout).	EXTI1	ADC1	INCAP1	ENC1
0	1 = Parameter error – out of range or invalid value.	EXTI0	ADC0	INCAP0	ENC0

CONFIGURE GPIO(s)

void **config_gpios(**uint8_t **iox_adrs**, uint8_t **start,** uint8_t **end,** uint8_t **gpio_config)**

Description

Configure one or more GPIO's with the same 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 GPIO's will default to analog input mode to minimize power consumption.

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 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 describing the type of I/O configuration requested.

The library header file contains macro constants which can be Ored together to create various configuration choices.

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

2. ***config_gpios(3, 19, 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.*

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 or ADC input).

OUTPUT 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

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).

Examples

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

2. ***config_gpios(3, 19, 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.*

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

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

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 bitwise map of all GPIO's to select for writing. A '1' bit associated with the logical GPIO will select that GPIO for writing (lower 20 bits).

'state_map' is a bitwise 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 gpio_map = 0x55555;          // write even bits with '1', odd bits with '0'

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

-------------------------------------------------------------------------

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

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

ret = write_gpios(0, gpio_map, gpio_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**, uint8_t **start,** uint8_t **end)**

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 the associated logical 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.

The **‘gpio_num’** 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**).

The state of the GPIO input is written to the memory location pointed to by the parameter **‘gpio_value’**.

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 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.

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. When an event occurs the GPIO output will go high and will be cleared to a low state by calling **READ STATUS**.

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 **config_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.

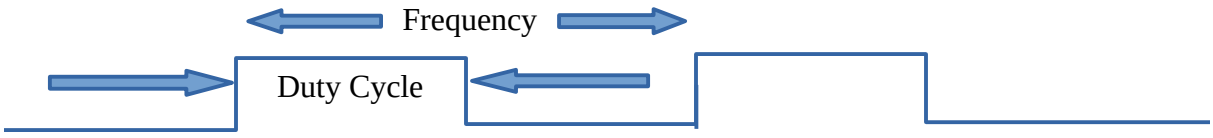
‘pwm_num’ *is t*he logical PWM pin (0 - 9) to output the desired PWM waveform.

‘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 asynchronously 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 results of ADC9 conversions

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

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 rising edge

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

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

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

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

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

The **READ STATUS** command with the field mask set to STATUS_ENCODER will return a map of encoder channels indicating which encoders were rotated.

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. This may not be accurate as encoders may have a different numbers of commutators.

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**(**3**,** 3**,** IO_INPUT | IO_PULLUP | IO_EXTI_FALLING**);** // wake on falling edge

ret = sleep(0, GPIO3); // 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.

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	Function successful
0x01	Parameter out of range.
0x02	General Failure
0x03	Serial bus mode error.

0x04	Communication timeout.

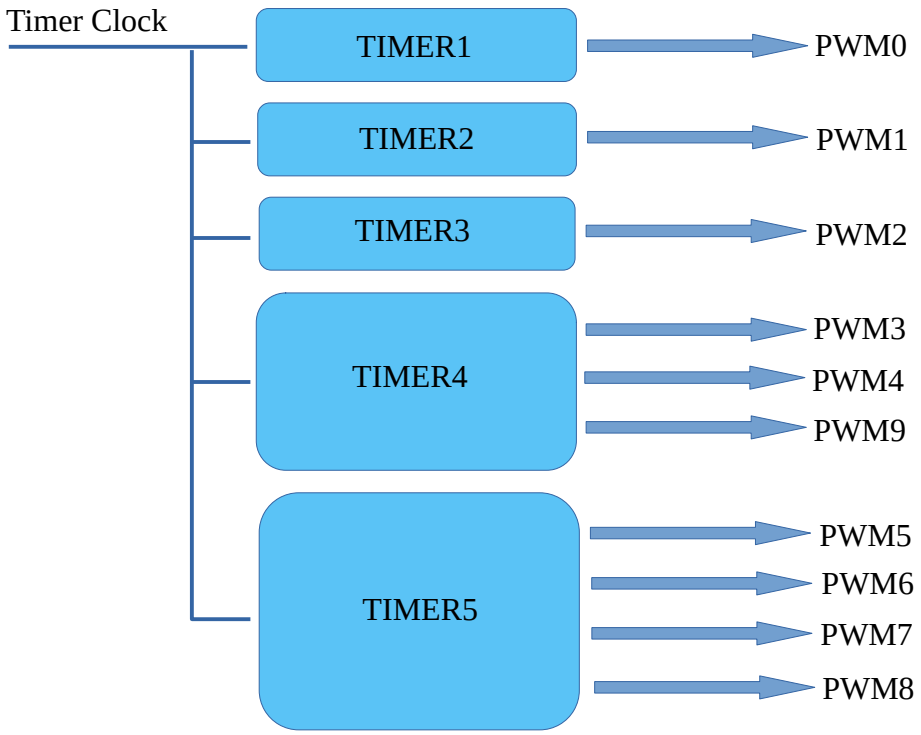
When an error occurs during serial bus communication, the library will return a bus failure error. The error is as follows:

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 MPU. 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-scale and period frequency values. Common period frequency is probably not an issue for servo control since multiple servos would require the same period frequency.



NOTES ON INPUT CAPTURE MEASUREMENTS

Input Capture measurements use timers embedded in the IOX MPU. 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.

Timers used for Input Capture cannot be used simultaneously 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 measurement interval to 89.48 seconds or 0.011 Hz.