BlueMod+S42 Lua API Documentation

30512ST10861A Rev. 1 – 2018-09-11

TELIT
TECHNICAL
DOCUMENTATION

SPECIFICATIONS ARE SUBJECT TO CHANGE WITHOUT NOTICE

NOTICES LIST

While reasonable efforts have been made to assure the accuracy of this document, Telit assumes no liability resulting from any inaccuracies or omissions in this document, or from use of the information obtained herein. The information in this document has been carefully checked and is believed to be reliable. However, no responsibility is assumed for inaccuracies or omissions. Telit reserves the right to make changes to any products described herein and reserves the right to revise this document and to make changes from time to time in content hereof with no obligation to notify any person of revisions or changes. Telit does not assume any liability arising out of the application or use of any product, software, or circuit described herein; neither does it convey license under its patent rights or the rights of others.

It is possible that this publication may contain references to, or information about Telit products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that Telit intends to announce such Telit products, programming, or services in your country.

COPYRIGHTS

This instruction manual and the Telit products described in this instruction manual may be, include or describe copyrighted Telit material, such as computer programs stored in semiconductor memories or other media. Laws in the Italy and other countries preserve for Telit and its licensors certain exclusive rights for copyrighted material, including the exclusive right to copy, reproduce in any form, distribute and make derivative works of the copyrighted material. Accordingly, any copyrighted material of Telit and its licensors contained herein or in the Telit products described in this instruction manual may not be copied, reproduced, distributed, merged or modified in any manner without the express written permission of Telit. Furthermore, the purchase of Telit products shall not be deemed to grant either directly or by implication, estoppel, or otherwise, any license under the copyrights, patents or patent applications of Telit, as arises by operation of law in the sale of a product.

COMPUTER SOFTWARE COPYRIGHTS

The Telit and 3rd Party supplied Software (SW) products described in this instruction manual may include copyrighted Telit and other 3rd Party supplied computer programs stored in semiconductor memories or other media. Laws in the Italy and other countries preserve for Telit and other 3rd Party supplied SW certain exclusive rights for copyrighted computer programs, including the exclusive right to copy or reproduce in any form the copyrighted computer program. Accordingly, any copyrighted Telit or other 3rd Party supplied SW computer programs contained in the Telit products described in this instruction manual may not be copied (reverse engineered) or reproduced in any manner without the express written permission of Telit or the 3rd Party SW supplier. Furthermore, the purchase of Telit products shall not be deemed to grant either directly or by implication, estoppel, or otherwise, any license under the copyrights, patents or patent applications of Telit or other 3rd Party supplied SW, except for the normal non-exclusive, royalty free license to use that arises by operation of law in the sale of a product.

USAGE AND DISCLOSURE RESTRICTIONS

I. License Agreements

The software described in this document is the property of Telit and its licensors. It is furnished by express license agreement only and may be used only in accordance with the terms of such an agreement.

II. Copyrighted Materials

Software and documentation are copyrighted materials. Making unauthorized copies is prohibited by law. No part of the software or documentation may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, without prior written permission of Telit.

III. High Risk Materials

Components, units, or third-party products used in the product described herein are NOT fault-tolerant and are NOT designed, manufactured, or intended for use as on-line control equipment in the following hazardous environments requiring fail-safe controls: the operation of Nuclear Facilities, Aircraft Navigation or Aircraft Communication Systems, Air Traffic Control, Life Support, or Weapons Systems (High Risk Activities"). Telit and its supplier(s) specifically disclaim any expressed or implied warranty of fitness for such High Risk Activities.

IV. Trademarks

TELIT and the Stylized T Logo are registered in Trademark Office. All other product or service names are the property of their respective owners.

V. Third Party Rights

The software may include Third Party Right software. In this case you agree to comply with all terms and conditions imposed on you in respect of such separate software. In addition to Third Party Terms, the disclaimer of warranty and limitation of liability provisions in this License shall apply to the Third Party Right software.

TELIT HEREBY DISCLAIMS ANY AND ALL WARRANTIES EXPRESS OR IMPLIED FROM ANY THIRD PARTIES REGARDING ANY SEPARATE FILES, ANY THIRD PARTY MATERIALS INCLUDED IN THE SOFTWARE, ANY THIRD PARTY MATERIALS FROM WHICH THE SOFTWARE IS DERIVED (COLLECTIVELY “OTHER CODE”), AND THE USE OF ANY OR ALL THE OTHER CODE IN CONNECTION WITH THE SOFTWARE, INCLUDING (WITHOUT LIMITATION) ANY WARRANTIES OF SATISFACTORY QUALITY OR FITNESS FOR A PARTICULAR PURPOSE.

NO THIRD PARTY LICENSORS OF OTHER CODE SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND WHETHER MADE UNDER CONTRACT, TORT OR OTHER LEGAL THEORY, ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE OTHER CODE OR THE EXERCISE OF ANY RIGHTS GRANTED UNDER EITHER OR BOTH THIS LICENSE AND THE LEGAL TERMS APPLICABLE TO ANY SEPARATE FILES, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

APPLICABILITY TABLE

- BLUEMOD+S42/AI/ADC/LUA

CONTENTS

1. INTRODUCTION

1.1. Scope

This document describes Lua libraries and their programming API used within the BlueMod+S42/ADC/Lua module.

1.2. Contact Information, Support

For general contact, technical support services, technical questions and report documentation errors contact Telit Technical Support at:

- TS-EMEA@telit.com
- TS-AMERICAS@telit.com
- TS-APAC@telit.com

or

- TS-SRD@telit.com for global Bluetooth support

Alternatively, use:

https://www.telit.com/contact-us/

For detailed information about where you can buy the Telit modules or for recommendations on accessories and components visit:

https://www.telit.com

Our aim is to make this guide as helpful as possible. Keep us informed of your comments and suggestions for improvements.

Telit appreciates feedback from the users of our information.

1.3. Text Conventions

Danger – This information MUST be followed or catastrophic equipment failure or bodily injury may occur.

Caution or Warning – Alerts the user to important points about integrating the module, if these points are not followed, the module and end user equipment may fail or malfunction.

Tip or Information – Provides advice and suggestions that may be useful when integrating the module.

All dates are in ISO 8601 format, i.e. YYYY-MM-DD.

1.4. Commonly Used Terms

ADC	Analog Digital Converter
API	Application Programming Interface
TBD	To be defined
TWI	Two Wire Interface

1.5. Related Documents

- [1] BlueMod+S42 Hardware User Guide, 1VV0301303
- [2] BlueMod+S42 Lua Software User Guide, 1VV0301471
- [3] Bluetooth 4.2 Core Specification
- [4] BlueMod+S42 LUA AT Command Reference, 80512ST10860A

2. OVERVIEW

This table shows the Lua packages that are available in addition to the standard Lua 5.3 packages. All of these packages are built into the firmware, but you have to use the `require()` function to make them available.

Package	Description
adc	Provides access to ADC channels.
crypto	Provides functions for cryptography.
dio	Provides access to digital I/O pins.
ev	Provides the ability to wait for events without busy-looping.
posix, posix.ctype, posix.dirent, posix.errno, posix.fcntl, posix.poll, posix.stdio, posix.sys.stat, posix.sys.statvfs, posix.termio, posix.time, posix.unistd	These packages provide a subset of functions and constants from the **LuaPOSIX** library that are implemented in the firmware.
spi	Provides master mode access to the serial-peripheral-interface bus.
target	Provides product specific functions and information.
twi	Provides master mode access to two-wire-interface bus.

Table 2-1 API Packages

For access to Bluetooth functionality like scanning for Beacons, configuring advertisement data or exchange data via GATT sevices or Terminal I/O profile please refer to [2] and [4].

Hard- and software features that can be well expressed as streams of bytes are made accessible through special device files. These files can be managed using the Lua standard IO functions like `io.read()` and `io.write()`, or if more control is needed, by **LUAPOSIX** library functions like `posix.unistd.read()` and `posix.unistd.write()`.

This table shows the device files available:

File	Description
/dev/null	Dummy file that always reads as empty and ignores writes.
/dev/zero	Dummy file that continually reads `"\0"` and ignores writes.
/dev/ttyS0	Provides access to the UART of the module.
/dev/ttyAT	A virtual comm port or modem that interprets AT commands for configuring the Bluetooth hardware.
/dev/ttyTIO0	Serves as an endpoint for a Terminal I/O connection.
/dev/mtdRO0	Provides access to a read-only area of the internal flash memory that contains the files in the /boot file system.
/dev/mtd0	Provides access to a 28 kbyte writable area of the internal flash memory that is typically mounted on /wo0.
/dev/mtd1	Provides access to a 4 kbyte writable area of the internal flash memory that is typically mounted on /wo1.
/dev/mtdSPI0	Can be configured to represent a SPI attached flash memory.
/dev/tim0	The timer counter used by `ev.create_tim("realtime")` internally.
/dev/tim1	The timer counter used by `ev.create_tim("monotonic")` internally.
/dev/dio	Represents the current state of the digital I/O pins. Internally used by event objects returned by `ev.create(dio.port1)` for monitoring pin state changes.
/dev/dioc	Represents the current configuration of the digital I/O pins.

Table 2-2 Device files

2.1. Analog Input Package: adc

The `adc` package contains functions for accessing the analog input pin of the module.

2.1.1. Function Table

Function	Description
config	Configures an analog to digital converter.
get	Triggers a conversion and returns the measured result.
persistent	Flags the ADC configuration to be kept active even when Lua exits.

Table 2-3 Analog Input Package API Functions

2.2. Cryptography Package: crypto

The `crypto` package contains functions for data encryption and verification.

2.2.1. Function Table

Function	Description
aes128	Encrypts a block of data with AES128.

Table 2-4 Cryptography Package API Functions

2.3. Digital I/O Package: dio

The `dio` package contains functions for accessing the digital I/O pins of the module.

2.3.1. Function Table

Function	Description
config	Configures digital I/O pins.
get	Reads state of digital I/O pins.
set	Writes state of digital I/O pins.
persistent	Flags the pin configurations to be kept active even when Lua exits.

Table 2-5 Digital I/O Package API Functions

2.4. Event Package: ev

The `ev` package contains functions that wait for specific events. While waiting for events, the module can save power by switching off some clocks, including the CPU clock.

2.4.1. Function Table

Function	Description
create	Creates an event object that can be waited for.
create_tim	Creates a timer event object that can be waited for.
wait	Waits for a single event object.
waitany	Waits for at least one event within a list of events.
waitall	Waits for all events within a list of events.
set	Sets the counter value of a timer event object.
get *(port event object)*	Reads state of digital I/O pins and updates the last known state in the event object.
get *(timer event object)*	Reads the counter value of a timer and updates the last known value in the event object.
settime	Sets expiration and interval times for a timer event object and returns the previous values.
gettime	Gets expiration and interval times for a timer event object.

Table 2-6 Event Package API Functions

2.4.2. Constant Table

Constant	Description
IN	1: Defines a flag for event objects that are triggered by the availability of input.
PRI	2: Defines a flag for event objects that are triggered by a condition depending on the purpose of the device file. For digital I/O ports, this condition would be complied if the state of the input pins has changed since the last read.
OUT	4: Defines a flag for event objects that are triggered by the availability of free output buffer space.
INOUT	5: Defines the bitwise OR combination of IN and OUT for event objects that are triggered by any of the two.
ERR	0x1000: Defines a flag for event objects that are triggered by the occurrence of an I/O error.
HUP	0x2000: Reserved. Not yet used by any event object.
NVAL	0x4000: Defines a flag for event objects that are triggered because they refer to a closed file.
INFINITE	-1: Defines the value for an infinite timeout.
ABSTIME	1: Defines a flag used to tell the settime() function that the given expiration time is absolute rather than relative to the current time.

Table 2-7 Event Package API Constants

2.5. Serial-Peripheral-Interface Package: spi

The `spi` package contains functions for accessing the serial-peripheral-interface bus of the module in master mode.

2.5.1. Function Table

Function	Description
config	Configures a SPI bus.
transfer	Transfers data to/from a connected SPI slave device.
persistent	Flags the bus configuration to be kept active even when Lua exits.
register_mtd	Registers a memory technology device (Flash/EEPROM) on the SPI bus and makes it accessible through a /dev/mtdSPI* file.
unregister_mtd	Undos the registration of a memory technology device on the SPI bus.

Table 2-8 Serial-Peripheral-Interface Package API Functions

2.6. Target Package: target

The `target` package contains functions for shutting down the device and information about the running firmware release.

2.6.1. Function Table

Function	Description
systemoff	Causes the device to shut down and enter lowest power consumption.
restart	Causes the device to shut down and restart.
mtd_info	Provides information about a memory technology device.
mtd_erase	Erases one sector/block of a memory technology device.
mtd_erase_all	Erases all sectors/blocks of a memory technology device.
mount	Mounts a file system.
umount	Unmounts a file system.

Table 2-9 Target Package API Functions

2.6.2. Constant Table

Constant	Description
FW	Firmware variant. (Example: “SBS2709”)
FW_VER	Full firmware version string. (Example: “SBS2709 V4.006 Jan 12 2018 16:17:08”)
FW_VER_MAJ	Major part of firmware version number. (Example: 4)
FW_VER_MIN	Minor part of firmware version number. (Example: 6)
LD_VER	Full boot loader version string. (Example: “SBS2703 V4.000”)
SD_FWID	0x9e: Nordic SoftDevice ID.
LF_CLK	Source of low frequency (32768Hz) clock: “internal” or “external”

Table 2-10 Target Package API Constants

2.7. Two-Wire-Interface Package: twi

The `twi` package contains functions for accessing the two-wire-interface bus of the module in master mode.

2.7.1. Function Table

Function	Description
config	Configures a TWI bus.
scan	Scans a TWI bus for connected slave devices.
transfer	Transfers data to/from a connected TWI slave device.
persistent	Flags the bus configuration to be kept active even when Lua exits.

Table 2-11 Two-Wire-Interface Package API Functions

3. FUNCTION DESCRIPTION

3.1. Analog Input Package Functions

3.1.1. adc.config

This function is used to get and set the configuration of an analog to digital converter.

Set configuration:

```
result = adc.config(converter, conf)
result = converter:config(conf)
```

May also return the triple `nil, errmsg, errcode`.

Get configuration:

```
conf = adc.config(converter)
conf = converter:config()
```

Parameter	Type	Description
converter	table	ADC whose configuration to get/set.
conf	table	ADC configuration to set. (see below)

Return value	Type	Description
result	boolean or nil	`true`, if successful; `nil`, if the configuration references a pin that is already used by another peripheral.
conf	table	Currently active ADC configuration. (see below)

The `converter` parameter is an opaque object that describes the analog to digital converter. `adc.adc1` is the only valid object for this parameter at the moment.

The `conf` parameter/return value is a table that may contain one entry for each configured channel. The key is the channel number in the range `1` to `4`. The value is another table that contains entries with the keys `SRC`, `GAIN`, `REF` and optionally `VDD`. Valid values for `SRC` are `"vdd"` (lets the channel measure the supply voltage) and integers in the range `1` to `8`, refering to the eight analog input pins of the chip. Valid values for `GAIN` are `1/6`, `1/5`, `1/4`, `1/3`, `1/2`, `1`, `2` and `4`. Valid values for `REF` are `"int"` and `"vdd/4"`. `"int"` refers to the internal reference of 0.6V and `"vdd/4"` refers to the supply voltage (VDD) divided by four. When `REF` is set to `"vdd/4"`, you also need to set `VDD` to your actual supply voltage in volts. This information is needed by the `adc.get()` function to calculate the voltage of the measurement result. When setting the configuration, only the channels that are indexed in the `conf` table are changed. All other channels keep their current settings. In order to disable a channel, set `SRC` to `nil`. When getting the configuration, the returned table contains only the channels that are configured. The result doesn't contain disabled channels.

Since the analog input pins may be shared with other peripherals (like the digital I/O port or the two-wire-interface), `adc.config()` will fail and return `nil, "No locks available", 46` if you have specified an analog input pin (`1`-`4`) that is already in use by another peripheral as the source (`SRC`) of any channel.

3.1.1.1. Example

```
-- Configure channel 1 to use analog input pin 1,
--                             internal reference (0.6V) and
--                             gain = 1/6
```

```
local conf = { [1] = { SRC = 1, REF = "int", GAIN = 1/6 } }
adc.adc1:config(conf)
-- measure on channel 1 and print the voltage
print("AIN pin: "..adc.adc1:get(1).."V")
-- disable channel 1
-- makes the pin avalaible for other peripherals
adc.adc1:config({ [1] = {} })
```

3.1.2. adc.get

This function is used to trigger a measurement on the given channel and return the result.

```
voltage, raw = adc.get(converter, channel)
voltage, raw = converter:get(channel)
```

Parameter	Type	Description
converter	table	ADC to use for conversion.
channel	integer	Channel to start the conversion on.

Return value	Type	Description
voltage	number or nil	Measured value of the analog input in volts.
raw	integer	Measured value of the analog input as raw 12bit value.

The `converter` parameter is an opaque object that describes the analog to digital converter. `adc.adc1` is the only valid object for this parameter at the moment.

If the given channel is disabled this function returns `nil`.

3.1.2.1. Example

```
-- Configure channel 1 to use analog input pin 1,
--                             external reference (VDD=3.3V/4) and
--                             gain = 1/4
-- Configure channel 2 to use supply voltage as input,
--                             internal reference (0.6V) and
--                             gain = 1/6
local conf = {
  [1] = { SRC = 1,     REF = "vdd/4", VDD = 3.3, GAIN = 1/4 },
  [2] = { SRC = "vdd", REF = "int",              GAIN = 1/6 }
}
adc.adc1:config(conf)
local ain_volt = adc.adc1:get(1) -- get pin voltage (based on VDD=3.3V)
local vdd_volt = adc.adc1:get(2) -- get supply voltage (VDD)
print("AIN pin: "..ain_volt.."V\nVDD: "..vdd_volt.."V")
```

```
-- disable channel 1&2
adc.adc1:config({ [1] = {}, [2] = {} })
```

3.1.3. adc.persistent

This function is used to configure the persistency of the ADC configuration.

```
adc.persistent(converter, state)
converter:persistent(state)
```

Parameter	Type	Description
converter	table	ADC whose persistency to configure.
state	boolean	`true`, if the channel configuration of `converter` should be kept alive even if Lua exits; `false`, if all channles should be disabled automatically on exit.

The `converter` parameter is an opaque object that describes the analog to digital converter. `adc.adc1` is the only valid object for this parameter at the moment.

After power up or reset, `state` is set to `false`, thus, channels get disabled automatically whenever a Lua script terminates. Though, this setting itself persists across script exits.

3.2. Cryptography Package Functions

3.2.1. crypto.aes128

Encrypts a block of data using AES128.

```
enc_data = crypto.aes128(data, key)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
data	string	A string with a length of 16 characters to encrypt.
key	string	A string with a length of 16 characters representing the key.

Return value	Type	Description
enc_data	string	A string with a length of 16 characters containing the encrypted data.

If the hardware AES peripheral is currently busy then `nil, "Device or resource busy", 16` is returned.

3.2.1.1. Example

```
local enc_data = crypto.aes128("foobar1234567890",
                               "\xde\xad\xca\xca1138\x42moo1234")
for i = 1, 16 do io.write(enc_data:byte(i).." ") end
-- prints: 50 77 10 64 238 106 88 238 2 186 197 93 164 248 98 136
```

3.3. Digital I/O Package Functions

3.3.1. dio.config

This function is used to get and set the configuration of the pins of a digital I/O port.

Set configuration:

```
result = dio.config(port, conf)
result = port:config(conf)
```

May also return the triple `nil, errmsg, errcode`.

Get configuration:

```
conf = dio.config(port)
conf = port:config()
```

Parameter	Type	Description
port	table	Port whose configuration to get/set.
conf	table	Pin configuration to set. (see below)

Return value	Type	Description
result	boolean or nil	`true`, if successful; `nil`, otherwise.
conf	table	Currently active pin configuration. (see below)

The `port` parameter is an opaque object that describes the port. `dio.port1` is the only valid object for this parameter at the moment.

The `conf` parameter/return value is a table that may contain one entry for each configured pin. The key is the pin number starting at 1. The value is another table that contains entries with the keys `DIR`, `PULL` and `FILTER`. Valid values for `DIR` are `"out"` and `"in"`. Valid values for `PULL` are `"pd"` and `"pu"`. `FILTER` may be set to an integer (0-75) that represents the time in milliseconds an input pin filters out glitches until a level change is considered stable. The default glitch filter time for input pins is 30 milliseconds. When setting the configuration, only the pins that are indexed in the `conf` table are changed. All other pins keep their current settings. In order to disable a pin, set `DIR` to `nil`. When getting the configuration, the returned table contains only the pins that are configured as either output or input. The result doesn't contain disabled pins.

Since the digital I/O pins may be shared with other peripherals (like the analog to digital converter or the two-wire-interface), `dio.config()` will fail and return `nil, "No locks available", 46` if you have tried to enable a pin that is already in use by another peripheral.

While the FILTER parameter identifies a time interval in milliseconds, the current status of an addressed input will be evaluated only every about 5ms. Due to that any signal change faster than this 5ms polling interval might be missed by this software mechanism and also any inter-input timing difference faster than this 5ms poll interval might be lost and the affected inputs changes might be signalled with one common event.

3.3.1.1. Example

```
local conf = {
  [1] = { DIR = "out" },       -- config pin 1 as output
  [3] = { DIR = "in" },        -- config pin 3 as input without pull
  [4] = { DIR = "in",          -- config pin 4 as input without pull
          FILTER = 10 },       -- with a 10ms degliching filter
  [8] = { DIR = "in",          -- config pin 8 as input with pull-down
          PULL = "pd" },       -- with pull-down
  [4] = {}                     -- disable pin 4
}
dio.port1:config(conf)         -- set configuration of port 1
```

3.3.2. dio.get

Returns the state of a pin or all pins of a port.

```
pin_states = dio.get(port)          or  pin_states = port:get()
pin_state = dio.get(port, pin_num)  or  pin_state = port:get(pin_num)
state, mask = dio.get(port, 0)      or  state, mask = port:get(0)
```

Parameter	Type	Description
port	table	Port whose pin state to get.
pin_num	integer	Pin number in range [1, 8].

Return value	Type	Description
pin_states	table	Table containing the state of all enabled pins of the port.
pin_state	boolean or nil	State of the pin `pin_num` of the port. `true` when the pin is high; `false` when it is low; and `nil` if it is disabled.
state	integer	A bitmap representing the state of all enabled pins of the port. 1 when a pin is high; 0 when a pin is low or disabled.
mask	integer	A bitmap representing all enabled pins of the port with 1. A pin is enabled when it is either configured as output or input.

The `port` parameter is an opaque object that describes the port. `dio.port1` is the only valid object for this parameter at the moment.

The `pin_states` return value is a table that contains one entry for each enabled pin of the port. If all pins are disabled, the table is empty. The key of an entry is the pin number [1, 8]. The value is the state of the pin. `true` when the pin is high; `false` when it is low.

3.3.3. dio.set

Sets the state of one or more output pins of a port.

```
dio.set(port, pin_states)           or  port:set(pin_states)
dio.set(port, pin_num, pin_state)   or  port:set(pin_num, pin_state)
dio.set(port, mask, state)          or  port:set(mask, state)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
port	table	Port whose pin state to set.
pin_num	integer	Pin number in range [1, 8].
pin_states	table	Table containing the new state of output pins.
pin_state	boolean	New state of the pin `pin_num` of the port. `true` sets the pin to high; `false` sets it to low.
state	integer	A bitmap representing the new state of all pins that are selected by `mask`. 1 sets a pin to high; 0 sets it to low. Pins not selected by `mask` are unchanged.
mask	integer	A bitmap representing all the output pins whose state should be set. 1 sets the state of the pin to the value of the respective bit in `state`; 0 leaves the pin unchanged.

The `port` parameter is an opaque object that describes the port. `dio.port1` is the only valid object for this parameter at the moment.

The `pin_states` parameter is a table that contains one entry for each output pin whose state should be set. The key of an entry is the pin number [1, 8]. The value is the new state of the pin. `true` when the pin is high; `false` when it is low. Pins that are not indexed in the table are unchanged. If the table is empty, the function call is a NOOP.

Trying to change the state of pins that are not configured as output causes this function to return `nil, "I/O error", 138`.

3.3.3.1. Example

```
local conf = { [2] = { DIR = "out" } } -- config pin 2 as output
dio.port1:config(conf)          -- set configuration of port 1
dio.port1:set(2, true)          -- set pin 2 to high
```

3.3.4. dio.persistent

This function is used to configure the persistency of the digital I/O configuration.

`dio.persistent``(port, state)`
`port:`**`persistent`**`(state)`

Parameter	Type	Description
port	table	Port whose persistency to configure.
state	boolean	`true`, if the pin configuration of `port` should be kept alive even if Lua exits; `false`, if all pins should be disabled automatically on exit.

The `port` parameter is an opaque object that describes the port. `dio.port1` is the only valid object for this parameter at the moment.

After power up or reset, `state` is set to `false`, thus, all pins get disabled automatically whenever a Lua script terminates. Though, this setting itself persists across script exits.

3.4. Event Package Functions

3.4.1. ev.create

Creates an event object from an open file or a digital I/O port. This event object can be used to wait for specific events of that file or port, like “input available” or “pin toggled”.

```
evobj = ev.create(file, flags)
evobj = ev.create(file)
evobj = ev.create(port)
```

Parameter	Type	Description
file	FILE*	An open file whose events should be made available by the event object.
port	table	Digital I/O port whose events should be made available by the event object.
flags	integer	Only when first parameter is a file: Flags that describe which events of the file should be signaled by the event object.

Return value	Type	Description
evobj	table	Event object that can be used to wait for the specified events of the file or port.

The `port` parameter is an opaque object that describes the port. `dio.port1` is the only valid object for this parameter at the moment.

Possible values for `flags` are: `ev.IN`, `ev.OUT`, `ev.PRI` or, the combination, `ev.INOUT`. I/O errors always cause the event object to get signaled, so the inclusion of the `ev.ERR` flag is not necessary.

If the first parameter is a file and `flags` is not given, then `ev.IN` is used for `flags`. If the first parameter is a port, the event object always uses the flag `ev.PRI`, which causes the event object to get signaled whenever the state of the pins change since the last time they have been read using `evobj:get()`. A digital I/O port can be read or written at all times without blocking, therefore waiting for `ev.IN` and `ev.OUT` would make no sense because they would always signal the event object immediately.

The `evobj` return value is an opaque object that provides a `wait()` method that can be used to wait for the occurrence of the event represented by the event object `evobj`. See `ev.wait()` for a description of that method. `evobj` can also be used by `ev.waitany()` and `ev.waitall()` if it is necessary to wait for events of multiple event objects.

If the port parameter is used, then `evobj` provides an additional method: `get()`.

3.4.1.1. Example

```
local uart_in_ev = ev.create(io.stdin)
uart_in_ev:wait() -- wait until there is data available to read from uart
local uart_out_ev = ev.create(io.stdin, ev.OUT)
uart_out_ev:wait() -- wait until there is free space in
                   -- the uart output buffer
```

3.4.2. ev.create_tim

Creates a timer event object. This timer event object can be used to wait for single shot or reoccurring timer events configured by `evobj:settime()`.

```
evobj = ev.create_tim(clockid)
evobj = ev.create_tim()
```

Parameter	Type	Description
clockid	string	Optional string that selects the clock source for the timer. `"monotonic"` and "`realtime`" are the only supported value for this parameter at the moment.

Return value	Type	Description
evobj	table	Event object that can be used to wait for a timer to expire.

The `evobj` return value is an opaque object that provides a `wait()` method that can be used to wait for the occurrence of the event represented by the event object `evobj`. See `ev.wait()` for a description of that method. `evobj` can also be used by `ev.waitany()` and `ev.waitall()` if it is necessary to wait for events of multiple event objects.

The `evobj` returned by this function also provides the methods: `set()`, `get()`, `settime()` and `gettime()`. The `get()` method returns the counter value of the timer and the `set()` method can be used to adjust this counter. The `settime()` method is used for setting the expiration and interval times of the timer. The current expiration time and interval can be read using the `gettime()` method.

3.4.2.1. Example

```
local tim = ev.create_tim()
tim:settime(5000) -- start timer; expire in five seconds
tim:wait()        -- wait for timer to expire
```

3.4.3. ev.wait

Wait for one event object to get signaled by the event it represents.

```
cause = ev.wait(evobj, timeout)
cause = ev.wait(evobj)
cause = evobj:wait(timeout)
cause = evobj:wait()
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
evobj	table	The event object that represents the event that should be waited for.
timeout	integer	Timeout in milliseconds after which the function should return even if the event doesn't occur.

Return value	Type	Description
cause	integer or boolean	If an event occurred, this is an integer that contains one or more of these flags: `ev.IN`, `ev.PRI`, `ev.OUT`, `ev.ERR`. `false`, if no event occurred within the timeout period.

The `evobj` parameter is an opaque object returned by `ev.create()` or `ev.create_tim()`.

If the `timeout` parameter is omitted or set to `ev.INFINITE`, the function never timeouts and therefore never returns a boolean value (`false`).

3.4.3.1. Example

```
local uart_ev = ev.create(io.stdin)
local cause = uart_ev:wait(5000)
if not cause then
  print "no char received within 5 seconds"
elseif (cause & ev.IN) == ev.IN then
  print "received something"
else -- (ev.ERR must be set)
  print "parity error? missing stop bit?"
end
```

3.4.4. ev.waitany

Wait for at least one event object to get signaled by the event it represents.

```
signaled, results = ev.waitany(list, timeout)
signaled, results = ev.waitany(list)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
list	table	A list with event objects that represent the events that should be waited for.
timeout	integer	Timeout in milliseconds after which the function should return even if none of the events occur.

Return value	Type	Description
signaled	boolean	`true`, if at least one event in `list` occurred; `false`, on timeout.
results	table or nil	If one or more event objects in `list` got signaled, this is a table that contains one entry for each of the signaled event objects. The key of each entry is the same key that the respective event object is assigned to in `list`. Therefore the key can be used to identify the event. The value of each entry is either an integer that contains one or more of these flags: `ev.IN`, `ev.PRI`, `ev.OUT`, `ev.ERR`; or `nil` if no event got signaled for the respective event object. `nil`, if none of the events occurred within the timeout period.

If the `timeout` parameter is omitted or set to `ev.INFINITE`, the function never timeouts and therefore never returns `false`.

If `list` is empty, the function waits until it timeouts, because it waits for *at least* one event. If, in this case, the timeout is not specified or `ev.INFINITE`, then the function blocks forever.

3.4.4.1. Example

```
local uart_ev = ev.create(io.stdin)
local pin_ev = ev.create(dio.port1)
local sig, result = ev.waitany({uart = uart_ev, pin = pin_ev}, 5000)
if not sig then
  print "no char received within 5 seconds and no pin change"
else
  if result.uart then
    if (result.uart & ev.IN) == ev.IN then
      print "received something"
    else -- (ev.ERR must be set)
      print "parity error? missing stop bit?"
    end
  end
  if result.pin then
    print "pin change"
  end
end
```

3.4.5. ev.waitall

Wait for all event objects to get signaled by the events they represent.

```
all, results = ev.waitall(list, timeout)
all, results = ev.waitall(list)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
list	table	A list with event objects that represent the events that should be waited for.
timeout	integer	Timeout in milliseconds after which the function should return even if not all of the events have occurred.

Return value	Type	Description
all	boolean	`true`, if all events in `list` occurred; `false`, on timeout.
results	table	This is a table that contains one entry for each of the signaled event objects. The key of each entry is the same key that the respective event object is assigned to in `list`. Therefore the key can be used to identify the event. The value of each entry is either an integer that contains one or more of these flags: `ev.IN`, `ev.PRI`, `ev.OUT`, `ev.ERR`; or `nil` if no event got signaled for the respective event object.

If the `timeout` parameter is omitted or set to `ev.INFINITE`, the function never timeouts and therefore never returns `false`.

If `list` is empty, the function returns immediately (`true, {}`), because the number of signaled events (zero) equals the number of events in `list` (also zero).

3.4.5.1. Example

```
local uart_ev = ev.create(io.stdin)
local pin_ev = ev.create(dio.port1)
local all, result = ev.waitall({uart = uart_ev, pin = pin_ev}, 5000)
if not all then
  print "none or only one of the two events occurred:"
  if result.uart then
    if (result.uart & ev.IN) == ev.IN then
      print "received something"
    else -- (ev.ERR must be set)
      print "parity error? missing stop bit?"
    end
  end
  if result.pin then
    print "pin change"
  end
else
  print "both events occurred"
end
```

3.4.6. evobj:set

Sets the counter value of a timer event object.

```
old_counter = evobj:set(new_counter)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
evobj	table	The timer event object whose counter value to set.
new_counter	integer	The new counter value to set for the timer.

Return value	Type	Description
old_counter	integer	The last known counter value of the timer that gets overwritten by `new_counter`.

The `evobj` parameter is an opaque object returned by `ev.create_tim()`.

The counter value of a new timer event object is initialized with 0. It gets incremented each time the timer expires or reaches its interval.

Setting the counter value while the timer is running is not recommended. The result of such an operation is undefined.

3.4.6.1. Example

```
local tim = ev.create_tim()
tim:set(1000)      -- set counter to 1000
tim:settime(5000) -- start timer; expire in five seconds
tim:wait()         -- wait for timer expiration
print(tim:get())   -- prints new counter value (1001)
```

3.4.7. evobj:get *(I/O port event object)*

Reads the state of a pin or all pins of a port and updates the last known state in the event object.

```
pin_states = evobj:get()
pin_state = evobj:get(pin_num)
state, mask = evobj:get(0)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
evobj	table	Port event object whose pin state to get.
pin_num	integer	Pin number in range [1, 8].

Return value	Type	Description
pin_states	table	Table containing the state of all enabled pins of the port.
pin_state	boolean or nil	State of the pin `pin_num` of the port. `true` when the pin is high; `false` when it is low; and `nil` if it is disabled.
state	integer	A bitmap representing the state of all enabled pins of the port. 1 when a pin is high; 0 when a pin is low or disabled.
mask	integer	A bitmap representing all enabled pins of the port with 1. A pin is enabled when it is either configured as output or input.

The `evobj` parameter is an opaque object returned by `ev.create()`.

The `pin_states` return value is a table that contains one entry for each enabled pin of the port. If all pins are disabled, the table is empty. The key of an entry is the pin number [1, 8]. The value is the state of the pin. `true` when the pin is high; `false` when it is low.

This method behaves like the `get()` method of the port (see `dio.get`), thus, it returns the pin state of the port, although with a significant side effect: It allows the event object insight about the last known pin state. This is important if you don't want to miss out on any pin state changes that happen between successive calls to `evobj:get()` and `ev.wait()`, `ev.waitany()` or `ev.waitall()`. To clarify this, consider the following code example:

```
dio.port1:config{ [1] = { DIR = "in" } } -- configure pin 1
local evobj = ev.create(dio.port1)
local button_pressed = evobj:get(1) -- get the state of a button
if not button_pressed then
  evobj:wait()                      -- wait for button to get pressed
end
```

If the button is not being pressed down when `evobj:get(1)` is called, then the script waits for it to get pressed by calling `evobj:wait()`. If the button gets pressed right after the call to `evobj:get(1)` but before the call to `evobj:wait()`, then `evobj:wait()` immediately returns without waiting, because the event object knows that the pin state has changed since the last call to `evobj:get()`. If you replace the `evobj:get()` call with `dio.port1:get(1)`, then the event object can't track the pin state change between the two calls. This means if the button gets pressed down between calling `dio.port1:get()` and `evobj:wait()`, then `evobj:wait()` blocks (until the pin state changes again). Please note that this example script is kept simple intentionally for illustrating the problem described here. In the real world,

`evobj:wait()` also returns if any other pin that is configured as input changes its state. So you should always put appropriate checks in place, for example, put a loop around the `evobj:wait()` that loops until pin 1 has really changed.

If an event object is used to detect input pin state changes of a digital I/O port, a changed signal must be stable for at the time interval defined by the DIO configuration parameter `FILTER` plus 11ms to be detected reliable. A non reliable debouncing mechanism is in place to reduce digital input state change detection due to signal glitches shorter than the time interval defined by the FILTER configuration.

3.4.8. evobj:get *(timer event object)*

Reads the counter of a timer and updates the last known value in the event object.

```
counter = evobj:get()
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
evobj	table	The timer event object whose counter value to read.

Return value	Type	Description
counter	integer	The counter value of the timer.

The `evobj` parameter is an opaque object returned by `ev.create_tim()`.

The counter value of a new timer event object is initialized with 0. It gets incremented each time the timer expires or reaches its interval.

3.4.8.1. Example

```
local tim = ev.create_tim()
local uart_ev = ev.create(io.stdin)
tim:settime(1000, 1000) -- start timer; counter increments each second
repeat
  local _, result = ev.waitany{uart = uart_ev, timer = tim}
  if result.timer then  -- counter incremented since last read?
    print(tim:get())    -- read and print counter value
  end
until result.uart       -- loop until uart received a byte
```

3.4.9. evobj:settime

Sets expiration and interval times for a timer event object.

```
old_expire, old_interval = evobj:settime(expire, interval, flags)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
evobj	table	The timer event object whose expiration and interval times to set.
expire	integer	The time in milliseconds the timer will expire in. The value 0 stops the timer.
interval	integer or nil	The interval in which the timer reoccurs. Use the value 0 or `nil` when starting a single shot timer.
flags	integer or nil	Flags that describe how to set up the timer. The flag `ev.ABSTIME` causes the `expire` parameter to be interpreted as an absolute point in time, rather than being interpreted relative to now.

Return value	Type	Description
old_expire	integer	The time in milliseconds the timer would have expired in.
old_interval	integer or none	The interval that was configured previously.

The `evobj` parameter is an opaque object returned by `ev.create_tim()`.

3.4.9.1. Example

```
local tim = ev.create_tim()
tim:settime(1000, 1000) -- start timer; timer event each second
-- ...
local e, i = tim:settime(0) -- stop timer
-- ...
if e == 0 then e = i end
tim:settime(e, i)           -- resume timer
```

3.4.10. evobj:gettime

Reads expiration and interval times of a timer event object.

```
expire, interval = evobj:gettime()
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
evobj	table	The timer event object whose expiration and interval times to get.

Return value	Type	Description
expire	integer	The time in milliseconds the timer will expired in.
interval	integer or none	The interval of the timer.

The `evobj` parameter is an opaque object returned by `ev.create_tim()`.

3.5. Serial-Peripheral-Interface Package Functions

3.5.1. spi.config

This function is used to configure and enable the serial-peripheral-interface controller.

Set configuration:

```
result = spi.config(bus, role, mode, clock, orc, bitorder)
result = bus:config(role, mode, clock, orc, bitorder)
```

May also return the triple `nil, errmsg, errcode`.

Get configuration:

```
role, mode, clock, orc, bitorder = spi.config(bus)
role, mode, clock, orc, bitorder = bus:config()
```

Parameter	Type	Description
bus	table	SPI bus whose configuration to get/set.
role	string	Role of the module on the SPI bus. `"master"` for master role; `""` for disabling SPI. The slave role is not supported.
mode	integer	SPI mode to use.
clock	integer	Clock to use on the SPI bus when configured as master. Optional. Defaults to 125000.
orc	string	Overread character to use. Optional. Defaults to `"\0"`.
bitorder	string	Bit order of all bytes on the bus. `"msb"` for most significant bit first. `"lsb"` for least significant bit first. Optional. Defaults to `"msb"`.

Return value	Type	Description
result	boolean or nil	`true`, if successful; `nil`, if the SCK, MOSI, MISO or SPI-CS pin is already used by another peripheral.
role	string	Role of the module on the SPI bus. `"master"` for master role; `""` if SPI is disabled.
mode	integer	SPI mode used. Only present when role is master.
clock	integer	Clock used on the SPI bus. Only present when role is master.
orc	string	Overread character. Only present when SPI is enabled.
bitorder	string	Bit order of all bytes on the bus. `"msb"` for most significant bit first. `"lsb"` for least significant bit first. Only present when SPI is enabled.

The `bus` parameter is an opaque object that describes the SPI bus. `spi.bus1` is the only valid object for this parameter at the moment.

The `mode` parameter defines the SPI mode (clock polarity and phase). Range 0-3.

The `clock` parameter defines how fast the bus gets clocked during a transfer. Valid values are `125000`, `250000`, `500000`, `1000000`, `2000000` and `4000000`.

The `orc` parameter defines the overread character that gets clocked out when you provide less data to send than you want to read. It is specified as an one character long string.

Since the pins of the serial-peripheral-interface may be shared with other peripherals (like the digital I/O port or the analog to digital converter), `spi.config()` will fail and return `nil, "No locks available", 46` if another peripheral already uses at least one of the SPI pins.

3.5.1.1. Example

```
spi.bus1:config("master", 0, 125000)  -- use Mode 0; 125kHz clock
spi.bus1:transfer("abc", 0)
spi.bus1:config("")                   -- disable SPI again
```

3.5.2. spi.transfer

Transfer data to/from a slave device connected to the SPI bus.

```
result = spi.transfer(bus, data_wr, length_rd)
result = bus:transfer(data_wr, length_rd)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
bus	table	SPI bus to use for transfer.
data_wr	string or nil	A string containing the data to be written to the slave device; or `nil`, if no data should be written.
length_rd	integer	Number of bytes to be read from the slave device. Valid range is 0 – 255.

Return value	Type	Description
result	string	A string containing the data read from the slave device. If `length_rd` is 0, this string is empty (`""`).

The `bus` parameter is an opaque object that describes the SPI bus. `spi.bus1` is the only valid object for this parameter at the moment.

3.5.2.1. Example

```
spi.bus1:config("master", 0, 125000) -- use mode 0; 125kHz clock

-- send write enable command (0x06)
spi.bus1:transfer("\x06", 0)
-- create program page string for SPI flash (command 0x02)
-- access memory address 0x000020 (encode as big endian 16bit ">H")
-- write data "somedata" (encode zero terminated string "z")
local data_wr = string.pack(">Hz", 0x03000020, "somedata")
spi.bus1:transfer(data_wr, 0)
```

```
-- create read data string for SPI flash (command 0x03)
-- access memory address 0x000020 (encode as big endian 16bit ">H")
data_wr = string.pack(">H", 0x03000020)
local data_rd = spi.bus1:transfer(data_wr, 12)
print(data_rd:sub(5)) -- prints "somedata", which was read back

spi.bus1:config("")                    -- disable SPI again
```

3.5.3. spi.persistent

This function is used to configure the persistency of the SPI bus configuration.

```
spi.persistent(bus, state)
bus:persistent(state)
```

Parameter	Type	Description
bus	table	SPI bus whose persistency to configure.
state	boolean	`true`, if the configuration of `bus` should be kept alive even if Lua exits; `false`, if the SPI bus should be disabled automatically on exit.

The `bus` parameter is an opaque object that describes the SPI bus. `spi.bus1` is the only valid object for this parameter at the moment.

After power up or reset, `state` is set to `false`, thus, the SPI bus gets disabled automatically whenever a Lua script terminates. Though, this setting itself persists across script exits.

3.5.4. spi.register_mtd

Registers a memory technology device (Flash/EEPROM) on the SPI bus and makes it accessible through a /dev/mtdSPI* file.

```
result = spi.register_mtd(bus, index, driver)
result = bus:register_mtd(index, driver)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
bus	table	SPI bus the device is attached to.
index	integer or nil	Selects the MTD file through which the device should be made accessible. `nil` selects the next available file that is not already mapped to a device. Use the return value to determine which file was choosen.
driver	string	Name of the driver that fits the device type.

Return value	Type	Description
result	integer	The integer suffix of a /dev/mtdSPI* file that provides access to the registered device.

The `bus` parameter is an opaque object that describes the SPI bus. `spi.bus1` is the only valid object for this parameter at the moment.

`"mx25"` is the only available driver. It supports Macronix MX25xxxx SPI Flash devices.

If `index` refers to a file that is already in use or if there is no unused file left, then `nil, "Device or resource busy", 16` is returned.

Once a device is registered and mapped to a file, this file can be opened and accessed like other files, except for write access: Due to the limitations of Flash memory, bits can't be flipped back to 1. Changing bits from 0 to 1 can only be achieved by erasing a whole sector. See `target.mtd_erase()` and `target.mtd_erase_all()`. /dev/mtdSPI* files can also be mounted using the Write-Once File System (WOFS) driver. See `target.mount()`.

3.5.4.1. Example

```
spi.bus1:config("master", 0, 4000000) -- use mode 0; 4MHz clock
spi.bus1:persistent(true)          -- do not disable SPI when Lua exits
spi.bus1:register_mtd(0, "mx25")       -- map flash to /dev/mtdSPI0
target.mount("/dev/mtdSPI0", "/spi")  -- mount on /spi
```

3.5.5. spi.unregister_mtd

Unregisters a memory technology device (Flash/EEPROM) from the SPI bus and invalidates the active file mapping.

```
result = spi.unregister_mtd(bus, index)
result = bus:unregister_mtd(index)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
bus	table	SPI bus the device is attached to.
index	integer	Selects the device by the integer suffix of the /dev/mtdSPI* file it was mapped to.

Return value	Type	Description
result	boolean or nil	`true`, if `index` referred to a mapped file.

The `bus` parameter is an opaque object that describes the SPI bus. `spi.bus1` is the only valid object for this parameter at the moment.

After unregistering a device, all further access to still open file handles will return the error: `nil, "Broken pipe", 32`

3.6. Target Package Functions

3.6.1. target.systemoff

This function causes the device to shut down and enter the lowest possible power consumption mode.

```
target.systemoff()
target.systemoff(mode)
```

Parameter	Type	Description
mode	string	Optional. `"wakeable"` for keeping input pins enabled to allow power on by digital I/O pin toggle. Otherwise, only the external RESET pin can power on the device.

All open files on non-volatile storage will get closed and synchronized during shutdown, after finalizer (`__gc`) functions got executed. When the device gets powered on again (either by RESET or digital I/O pin), it goes through a full system reset, no RAM contents and no digital I/O configurations are preserved.

For being able to wake the device by digital I/O pins, in addition to passing `"wakeable"` for the `mode` parameter, you also need to make sure the configuration of the I/O port is kept alive when Lua shuts down. See `dio.persistent()` and/or the example below.

3.6.1.1.1. Example

```
local conf = {
  [8] = { DIR = "in", PULL = "pd" },
}
dio.port1:config(conf)        -- set configuration of port 1
dio.port1:persistent(true)    -- keep configuration of port 1 alive
target.systemoff("wakeable") -- can be powered on by RESET or DIO8
```

3.6.2. target.restart

This function causes the device to shut down and restart.

```
target.restart()
target.restart(mode, interface, devname)
```

Parameter	Type	Description
mode	string	Optional. `"firmware"` (default) for restarting normally. `"dfu"` for entering device firmware update mode after restart.
interface	string	Only for DFU; optional. `"serial"` (default) for updating over UART. `"ble"` for updating over the air (OTA) using Bluetooth Low-Energy.
devname	string	Only for OTA. Device name the device should use to advertise itself.

All open files on non-volatile storage will get closed and synchronized during shutdown, after finalizer (`__gc`) functions got executed.

3.6.3. target.mtd_info

Inquires information about a memory technology device.

```
info = target.mtd_info(dev_file)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
dev_file	FILE* or integer	A Lua file object or a file descriptor referring to an open MTD file that is mapped to the device that should be inquired.

Return value	Type	Description
info	table	Table with the following keys containing information about the device: `"device_size"`: Size of the device in bytes `"erase_size"`: Size of a sector in bytes. Bytes can only be erased in a union of this quantity. `"read_only"`: `true`, if the device is read-only; otherwise, `false`. `"has_cache"`: `true`, if the device has a r/w cache; otherwise, `false`.

3.6.4. target.mtd_erase

Erase one sector of a memory technology device.

```
result = target.mtd_erase(dev_file)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
dev_file	FILE* or integer	A Lua file object or a file descriptor referring to an open MTD file that is mapped to the device of which one sector should be erased.

Return value	Type	Description
result	boolean or nil	`true`, if the sector was erased successfully.

The current r/w position of the file selects which sector will be erased. It can point at any byte within a sector. For example, if the file position points at byte 4100 and the sector size of the device is 4096, then the second sector (that is sector #1 if counting starts at 0) will be erased. Any position from 0-4095 inclusively would erase the first sector, any position from 4096-8191 would erase the second one, etc.

When the file position points beyond the end of the device, `nil, "No space left on device", 28` is returned.

When the sector was erased successfully, the file position is advanced to the first byte within the next sector. Therefore, calling `target.mtd_erase()` three times in a row would erase three consecutive sectors.

3.6.4.1. Example

```
target.umount("/wo0") -- can't write/erase while device is mounted
local f = io.open("/dev/mtd0", "r+b") -- open device for r/w
local inf = target.mtd_info(f)     -- get sector size (spoiler: 4096)
f:seek("set", inf.erase_size * 2) -- position file at 8192
target.mtd_erase(f)                -- erases 3rd sector
target.mtd_erase(f)                -- erases 4th sector
f:close()
```

3.6.5. target.mtd_erase_all

Erase all sectors of a memory technology device.

```
result = target.mtd_erase_all(dev_file)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
dev_file	FILE* or integer	A Lua file object or a file descriptor referring to an open MTD file that is mapped to the device which should be fully erased.

Return value	Type	Description
result	boolean or nil	`true`, if the device was erased successfully.

The current r/w position of the file is irrelevant. The erase operation is started even if the file position points beyond the end of the device.

When all sectors were erased successfully, the file position is rewinded to 0.

3.6.5.1. Example

```
target.umount("/wo0") -- can't write/erase while device is mounted
local f = io.open("/dev/mtd0", "r+b") -- open device for r/w
target.mtd_erase_all(f)                -- erase whole device
f:close()
target.mount("/dev/mtd0", "/wo0")      -- mount again
-- Note: This is essentially what the shell command >wofmt< does.
```

3.6.6. target.mount

Mounts a file system.

```
result = target.mount(source, target, fstype, data)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
source	string	The device file representing the underlying storage medium for the file system.
target	string	The mount point. The path through which the file system contents should be made available.
fstype	string or nil	Optional. The name of the file system driver being used. Defaults to `"romfs"`.
data	string or nil	Optional. Additional data that gets passed to the file system driver. Defaults to `""`. Interpreted as a comma separated list of options like `"ro"`, `"rw"` or `"remount"`.

Return value	Type	Description
result	boolean or nil	`true`, if the file system was mounted successfully.

List of file system names that can be given for the `fstype` parameter:

`"romfs"`: Default. A simple file system for read-only memory. It can also operate as a write-once file system (WOFS) if the underlying device is writable. This is the file system /wo0 and /wo1 are mounted with.

`"ramfs"`: A file system that exists solely in volatile RAM. The contents vanish on restart or power loss. It does not use a storage medium so the `source` parameter is ignored.

List of common options for the `data` parameter that are respected by all file systems:

`"rw"`: Default. Mount file system writable. This will fail for devices that are read-only, like /dev/mtdRO0.

`"ro"`: Mount file system read-only.

`"remount"`: Remount an already mounted file system with different options without unmounting it in between. Currently open files don't need to be closed, as long as their access modes (r/w) don't violate the new mount options. The `source` and `fstype` parameters are ignored when remounting a file system.

3.6.6.1. Example

```
target.mount("", "/wo1", nil, "remount,ro") -- remount /wo1 read-only
```

3.6.7. target.umount

Unmounts a file system.

```
result = target.umount(target)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
target	string	The mount point of the file system.

Return value	Type	Description
result	boolean or nil	`true`, if the file system was unmounted.

3.7. Two-Wire-Interface Package Functions

3.7.1. twi.config

This function is used to configure and enable the two-wire-interface controller.

Set configuration:

```
result = twi.config(bus, role, clock)
result = bus:config(role, clock)
```

May also return the triple `nil, errmsg, errcode`.

Get configuration:

```
role, clock = twi.config(bus)
role, clock = bus:config()
```

Parameter	Type	Description
bus	table	TWI bus whose configuration to get/set.
role	string	Role of the module on the TWI bus. `"master"` for master role; `""` for disabling TWI. The slave role is not supported.
clock	integer	SCL clock to use on the TWI bus when configured as master. Optional. Defaults to 100000.

Return value	Type	Description
result	boolean or nil	`true`, if successful; `nil`, if the SDA or SCL pin is already used by another peripheral.
role	string	Role of the module on the TWI bus. `"master"` for master role; `""` if TWI is disabled.
clock	integer	SCL clock used on the TWI bus. Only present when role is master.

The `bus` parameter is an opaque object that describes the TWI bus. `twi.bus1` is the only valid object for this parameter at the moment.

The `clock` parameter defines how fast the bus gets clocked during a transfer. Valid values are `100000`, `250000` and `400000`.

Since the pins of the two-wire-interface may be shared with other peripherals (like the digital I/O port or the analog to digital converter), `twi.config()` will fail and return `nil, "No locks available", 46` if another peripheral already uses at least one of the TWI pins.

3.7.1.1. Example

```
twi.bus1:config("master", 100000)  -- use 100kHz SCL clock
twi.bus1:transfer(0x10, "abc", 0)
twi.bus1:config("")                -- disable TWI again
```

3.7.2. twi.scan

Scan the TWI bus for connected slave devices.

```
devices = twi.scan(bus)
devices = bus:scan()
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
bus	table	TWI bus to scan for slave devices.

Return value	Type	Description
devices	table	A list of device addresses (integer) that responded with an ACK on the TWI bus.

The `bus` parameter is an opaque object that describes the TWI bus. `twi.bus1` is the only valid object for this parameter at the moment.

The `twi.scan()` function tries to reach all slave addresses on the TWI bus and returns a list of device addresses that responded with an ACK.

If the TWI bus is not in IDLE state (another TWI master is using it, or a TWI slave is permanently clock stretching), the error triple `nil, "Remote I/O error", 145` is returned.

3.7.2.1. Example

```
twi.bus1:config("master", 100000)         -- use 100kHz SCL clock
local devices = twi.bus1:scan()
for _, addr in ipairs(devices) do         -- print out all addresses
  print(addr)
end
twi.bus1:config("")                       -- disable TWI again
```

3.7.3. twi.transfer

Transfer data to/from a slave device connected to the TWI bus.

```
result = twi.transfer(bus, address, data_wr, length_rd)
result = bus:transfer(address, data_wr, length_rd)
```

May also return the triple `nil, errmsg, errcode`.

Parameter	Type	Description
bus	table	TWI bus to use for transfer.
address	integer	Device address of the TWI slave device. Valid range is 0x00 – 0x7F.
data_wr	string or nil	A string containing the data to be written to the slave device; or `nil`, if no data should be written.
length_rd	integer	Number of bytes to be read from the slave device. Valid range is 0 – 255.

Return value	Type	Description
result	string or boolean	`false`, if a NAK was read back from the bus after writing the slave address or the data. Otherwise, a string containing the data read from the slave device. If `length_rd` is 0, this string is empty (`""`).

The `bus` parameter is an opaque object that describes the TWI bus. `twi.bus1` is the only valid object for this parameter at the moment.

The `twi.transfer()` function executes a single write, single read or a combined write/read TWI transaction to/from a TWI slave device.

If the `data_wr` parameter is not `nil`, the `twi.transfer()` function sends a START condition followed by a write access to the slave address on the TWI bus. Then the content of the the `data_wr` parameter is written to the device, followed by a STOP condition if the `length_rd` parameter is 0. If the `length_rd` parameter is not 0, a REPEATED START condition is put on the TWI bus instead, followed by a read access from the slave address. Then `length_rd` bytes are read from the slave device, followed by a STOP condition.

If the `data_wr` parameter is `nil` and the `length_rd` parameter is not 0, a START condition followed by a read access from the slave address is put on the TWI bus. Then `length_rd` bytes are read from the slave device, followed by a STOP condition.

If the `data_wr` parameter is `nil` and the `length_rd` parameter is 0, this function behaves like `data_wr` is an empty string; thus sending a START condition, followed by a write access to the slave address with no data, followed by a STOP condition.

If a NAK was read back from the bus directly after the slave address was written (address NAK), `false` is returned by this function. If a NAK was read back during the data write phase, after a successfully ACKed slave address (data NAK), two values are returned: `false, true`.

If the TWI bus is not in IDLE state (another TWI master is using it, or a TWI slave is permanently clock stretching), the error triple `nil, "Remote I/O error", 145` is returned.

3.7.3.1. Example

```
twi.bus1:config("master", 100000)     -- use 100kHz SCL clock

-- create write data string for TWI eeprom (bus address 0x50)
-- access memory address 0x0020 (encode as big endian 16bit ">H")
-- write data "somedata" (encode zero terminated string "z")
local data_wr = string.pack(">Hz", 0x0020, "somedata")
twi.bus1:transfer(0x50, data_wr, 0)

-- create read data string for TWI eeprom (bus address 0x50)
-- access memory address 0x0020 (encode as big endian 16bit ">H")
data_wr = string.pack(">H", 0x0020)
local data_rd = twi.bus1:transfer(0x50, data_wr, 8)
print(data_rd) -- prints "somedata", which was read back from eeprom

twi.bus1:config("")                   -- disable TWI again
```

3.7.4. twi.persistent

This function is used to configure the persistency of the TWI bus configuration.

```
twi.persistent(bus, state)
bus:persistent(state)
```

Parameter	Type	Description
bus	table	TWI bus whose persistency to configure.
state	boolean	`true`, if the configuration of `bus` should be kept alive even if Lua exits; `false`, if the TWI bus should be disabled automatically on exit.

The `bus` parameter is an opaque object that describes the TWI bus. `twi.bus1` is the only valid object for this parameter at the moment.

After power up or reset, `state` is set to `false`, thus, the TWI bus gets disabled automatically whenever a Lua script terminates. Though, this setting itself persists across script exits.

4. DEVICE FILE DESCRIPTION

This section describes which I/O operations can be used on the special device files. For each file there will be a table that gives a quick overview about the provided operations:

read: yes	write: no	seek: dummy	poll: [IN]/-/PRI/ERR	ioctl: no

If an operation is not supported (no), the matching Lua function will return an error code. Dummy means the function does not return an error, but it does not do anything useful either.

The columns “read”, “write” and “seek” reflect if the functions

`io.read()`, `io.write()` and `io.seek()`

or `posix.unistd.read()`, `posix.unistd.write()` and `posix.unistd.lseek()`

can be used respectively.

The column “poll” reflects if the function `posix.poll.poll()` can be used to wait for events generated by the file. There are three “non-error” events: IN - the file has at least one byte available to be read; OUT - the file is able to accept at least one byte to be written to it; PRI - some other type of event that is special to the purpose of the file occurred. The IN event is also set if the next read operation would signal the end of the file. End-of-file causes the read operation to return without blocking, therefore it is a valid source for this event. If the event is set in square brackets ([IN]), this means the file is always ready for read or write respectively, therefore polling for that event would be useless. Some files report the error event (ERR).

The column “ioctl” reflects if the file provides additional operations that are not covered by standard file I/O. There is no generic Ioctl function in Lua. Read the description for the file to learn how the additional operations of that particular file are accessed in Lua.

4.1. /dev/null

read: yes	write: yes	seek: dummy	poll: [IN]/[OUT]/-/-	ioctl: no

Always returns end-of-file when read. Writes are always accepted but all data is ignored.

Can be used in shell scripts to redirect unwanted output from STDOUT or STDERR. The module actually starts up with STDIO directed to this file until the UART gets configured.

4.2. /dev/zero

read: yes	write: yes	seek: dummy	poll: [IN]/[OUT]/-/-	ioctl: no

Always returns the requested amount of bytes when read (filled with NUL bytes: `"\0"`). Writes are always accepted but all data is ignored.

A cheap endless source of known data. Can be used to benchmark file based data throughput.

4.3. /dev/ttyS0

read: yes	write: yes	seek: dummy	poll: IN/OUT/-/ERR	ioctl: yes

Provides access to the UART. Reading this file returns the requested amount of bytes from the RX buffer (if available) and removes them from the buffer. Writing enqueues data to the TX buffer if space is available.

If the RX buffer is empty, a read operation blocks until at least one byte got received. If opened with the O_NONBLOCK flag, the EAGAIN error code is returned instead of blocking.

When the UART is disabled, the read operation signals end-of-file instead of waiting for a new byte being received.

If the TX buffer can't accept any more data, a write operation blocks until at least one byte got written to the buffer. If opened with the O_NONBLOCK flag, the EAGAIN error code is returned instead of blocking.

If opened with the O_SYNC flag, write operations additionally block until all bytes from the TX buffer were written into the hardware TX shift register or an error occurs while doing so. This has precedence over the O_NONBLOCK flag, but only if new data was actually written to the TX buffer by this operation.

Write operations that get issued while the UART is disabled terminate immediately (returning 0), even with the O_SYNC flag set. The OUT event won't get signaled for a disabled UART. Data that is already in the TX buffer when the UART gets disabled will get transmitted before it actually disables the transmitter.

The UART device file supports IOCTLs common to TTY devices. The `posix.termio` package provides functions that utilize these IOCTLs:

- `posix.termio.tcflush()` flushes the RX and/or TX buffer, discarding unread or unsent data.
- `posix.termio.tcdrain()` drains the TX buffer; blocks until all data got written into the TX shift register.
- `posix.termio.tcgetattr()` reads the current configuration of the device.
- `posix.termio.tcsetattr()` configures the device. If the baudrate, parity or flow control is changed, the TX buffer gets drained and the RX buffer gets flushed before the changes are effective.

The UART doesn't support separate baudrates for RX and TX. The `ispeed` and `ospeed` fields in the table returned by `tcgetattr()` are always identical. `tcsetattr()` ignores `ispeed` if both fields are set.

If the baudrate is set to 0 or `posix.termio.B0`, the UART gets disabled. This reduces power consumption.

If set, the following TTY flags have an effect:

cflag: CRTSCTS**	Use hardware flow control (RTS/CTS).
cflag: PARENB	Use parity bit. Only even parity is supported.
lflag: ICANON*	Read operations return EOF if character 4 <EOT> (CTRL+D) is received.
lflag: ECHO**	All characters that get read are automatically echoed. (Echoed characters are processed according to iflags and oflags.)

lflag: ECHONL	(Implied if ECHO is set.) Newlines that get read are automatically echoed. Character 10 <LF> is interpreted as newline, but iflags are applied first; oflags are applied afterwards.
lflag: ECHOCTL**	(Ignored if ECHO is not set.) Non-printable characters are echoed in printable form. All control characters (below 32) except <TAB> and <LF> are considered non-printable. For example, character 4 <EOT> is echoed as “^D”; character 0 <NUL> is echoed as “^@”; character 13 <CR> is echoed as “^M”.
iflag: ISTRIP	MSB is ignored on read; all read characters are masked with 0x7f.
iflag: INLCR	Map character 10 <LF> to 13 <CR> on read.
iflag: IGNCR	Ignore character 13 <CR> on read.
iflag: ICRNL*	(Ignored if IGNCR is also set.) Map character 13 <CR> to 10 <LF> on read.
iflag: IUCLC	Map upper case characters to lower case on read. [A-Z]->[a-z]
oflag: OPOST*	If not set, all other oflags are ignored.
oflag: ONLCR*	(Ignored if OPOST is not set.) Prefix each character 10 <LF> with 13 <CR> on write, effectively transmitting (Win)DOS line endings <CR><LF>.
oflag: OCRNL	(Ignored if OPOST is not set.) Map character 13 <CR> to 10 <LF> on write.
oflag: ONLRET*	(Ignored if OPOST is not set.) Don’t output character 13 <CR>. ONLCR is not affected by this flag.
oflag: OLCUC	(Ignored if OPOST is not set.) Map lower case characters to upper case on write. [a-z]->[A-Z]

*: On startup, the UART is disabled (baudrate = 0) and these flags are set.

**: These flags are additionally set when the startup script /boot/init.sh is used to prepare the UART as a terminal for an interactive shell. Also a baudrate of 115200 is configured, which enables the UART.

4.4. /dev/ttyAT

read: yes	write: yes	seek: dummy	poll: IN/OUT/-/ERR	ioctl: yes

AT commands can be written to this virtual comm port to communicate with the Bluetooth Low Energy subsystem of the module. Command responses and asynchronous events can be read.

If the RX buffer is empty, a read operation blocks until the AT handler enqueues the response for a command that got issued prior to calling read() or if it enqueues an asynchronous BLE event that has occurred. If opened with the O_NONBLOCK flag, the EAGAIN error code is returned instead of blocking.

If the TX buffer can't accept any more data, a write operation blocks until at least one byte got written to the buffer. If opened with the O_NONBLOCK flag, the EAGAIN error code is returned instead of blocking.

If opened with the O_SYNC flag, write operations additionally block until all bytes from the TX buffer got fetched by the AT handler or an error occurs. This has precedence over the O_NONBLOCK flag, but only if new data was actually written to the TX buffer by this operation.

The ERR event gets signaled if the RX buffer overruns. The buffer must be read or flushed to clear the error.

Closing this file results in a reset of the Bluetooth Low Energy subsystem. All file handles referencing the AT handler device file must be closed for this reset to take place.

The AT handler device file supports IOCTLs common to TTY devices. The `posix.termio` package provides functions that utilize these IOCTLs:

- `posix.termio.tcflush()` flushes the RX and/or TX buffer, discarding unread responses/events or unprocessed commands.
- `posix.termio.tcdrain()` drains the TX buffer; blocks until all commands got fetched by the AT handler.
- `posix.termio.tcgetattr()` reads the current configuration of the device.
- `posix.termio.tcsetattr()` configures the device.

Baudrate, parity and handshaking configuration has no effect on this device.

The following TTY flags are set by default for the AT handler: IGNCR, OPOST and ONLCR.

The AT handler uses character 13 <CR> as newline plus an optional 10 <LF>. The IGNCR and ONLCR flags (set by default) map this line ending to the one Lua uses: 10 <LF>. This way, standard I/O functions can be used, for example, `io.read("*l")` for reading a line.

If set, the flags listed for /dev/ttyS0 (except the cflags) have the same effect on this device file. Although some may be kind of fatal, like ECHO, which would write everything read from the AT handler back to it. (Combine that with ATE1!)

4.5. /dev/ttyTIO0

read: yes	write: yes	seek: dummy	poll: IN/OUT/-/-	ioctl: yes

Acts as an endpoint of a Telit Terminal I/O connection. Reading this file returns the requested amount of bytes from the RX buffer (if available) and removes them from the buffer. Writing enqueues data to the TX buffer if space is available.

If the RX buffer is empty, a read operation blocks until at least one byte is received. If opened with the O_NONBLOCK flag, the EAGAIN error code is returned instead of blocking.

When the endpoint is disconnected and the RX buffer is empty, the read operation signals end-of-file instead of waiting for a new byte being received. If there is still data in the RX buffer, the contents of the buffer are read first before signaling the end-of-file. If the remote device re-establishes a connection, the local endpoint is guaranteed to signal end-of-file at least once after the data from the previous connection got read completely (or flushed by the application), even if read() is called after the new connection got established again, therefore allowing the application to synchronously detect that there was a disconnect in between.

If the TX buffer can't accept any more data, a write operation blocks until at least one byte got written to the buffer. If opened with the O_NONBLOCK flag, the EAGAIN error code is returned instead of blocking.

If opened with the O_SYNC flag, write operations additionally block until all bytes from the TX buffer got fetched by the Terminal I/O profile handler of the Bluetooth Low Energy subsystem or an error occurs. This has precedence over the O_NONBLOCK flag, but only if new data was actually written to the TX buffer by this operation.

Write operations that get issued or are ongoing/blocking when the endpoint is disconnected terminate immediately (returning the EPIPE error code), even with the O_SYNC flag set. The OUT event won't get signaled for a disconnected endpoint. Data that is already in the TX buffer when the endpoint gets disconnected is discarded (it won't get carried over to a new connection).

The Terminal I/O endpoint device file supports IOCTLs common to TTY devices. The `posix.termio` package provides functions that utilize these IOCTLs:

- `posix.termio.tcflush()` flushes the RX and/or TX buffer, discarding unread or unsent data.
- `posix.termio.tcdrain()` drains the TX buffer; blocks until all data got fetched by the Terminal I/O profile handler of the Bluetooth Low Energy subsystem.
- `posix.termio.tcgetattr()` reads the current configuration of the device.
- `posix.termio.tcsetattr()` configures the device.

Baudrate, parity and handshaking configuration has no effect on this device.

All TTY flags are cleared by default. If set, the flags listet for /dev/ttyS0 have the same effect on this device file (except for the cflags).

4.6. /dev/mtdRO0

read: yes	write: no	seek: yes	poll: [IN]/[OUT]/-/-	ioctl: yes

Provides read access to the internal flash area that gets mounted on /boot.

The device file can be read bytewise and seeking is possible. When the file position points beyond the size of the device, read signals end-of-file.

This read-only memory technology device file supports one IOCTL accessible by the following function from the `target` package:

- `target.mtd_info()` provides information about the device, like its size.

4.7. /dev/mtd0

read: yes	write: yes	seek: yes	poll: [IN]/[OUT]/-/-	ioctl: yes

Provides read and write access to the internal flash area that gets mounted on /wo0.

The device file can be read and written bytewise and seeking is possible. When the file position points beyond the size of the device, read signals end-of-file and write returns the error code ENOSPC (no space left on device).

It is allowed to overwrite already written bytes without erasing the whole sector first, though zero bits can not be set to 1 this way. All write operations are verified. If the overwritten byte already contained zero bits that were not zero in the buffer provided to the write operation, then the operation returns the EIO error code and the file position will not get advanced beyond that byte. (The exact position the file is left with is not defined in case of any error.)

This device file implements a single row 16 byte cache for read and write operations. The row is always aligned at 16 bytes. Consecutive write operations that fit into the same 16 byte row are delayed until one of the following occurs: A write operation gets issued with a destination located outsite the 16 byte boundary; The device file gets closed; `io.flush()` gets called. If the sector or the whole device gets erased, the cached row won't be written at all. The cache is transparent for the application: Read operations consult the cache first and the cache imitates the behavior of the underlying flash memory when it comes to overwriting bytes. Thus, errors that are the result of trying to overwrite zero bits with ones are reported synchronously as described in the paragraph above, even when the actual write to flash memory is delayed.

The memory technology device files support IOCTLs accessible by the following functions from the `target` package:

- `target.mtd_info()` provides information about the device, like its size and the size of the erasable sectors.
- `target.mtd_erase()` erases the sector pointed to by the current file position.
- `target.mtd_erase_all()` erases the whole device.

If the device is mounted by a file system driver, write and erase operations on this file will be rejected and return the error code ETXTBSY.

4.8. /dev/mtd1

read: yes	write: yes	seek: yes	poll: [IN]/[OUT]/-/-	ioctl: yes

Provides read and write access to the internal flash area that gets mounted on /wo1.

See the description of /dev/mtd0 for more details.

4.9. /dev/mtdSPI0

read: yes	write: yes	seek: yes	poll: [IN]/[OUT]/-/-	ioctl: yes

Provides read and write access to an external memory attached to a SPI bus.

The function `spi.register_mtd()` can be used to map this device file to a memory device attached to a SPI bus. Accessing this device file while it is unmapped will return the error code EPIPE.

See the description of /dev/mtd0 for more details.

4.10. /dev/tim0

read: yes	write: no	seek: yes	poll: IN/-/-/-	ioctl: yes

This device file is internally used by event objects returned by `ev.create_tim("realtime")`.

The IOCTLs necessary to utilize this device file in a useful way are not directly exposed as Lua functions.

The clock used by this device can be set by `posix.time.clock_settime()`. Therefore timer/counters based on this device could do huge jumps when the clock gets set. It is useful for timer/counters that generate events at absolute wall clock times.

4.11. /dev/tim1

read: yes	write: no	seek: yes	poll: IN/-/-/-	ioctl: yes

This device file is internally used by event objects returned by `ev.create_tim("monotonic")`.

The IOCTLs necessary to utilize this device file in a useful way are not directly exposed as Lua functions.

The clock used by this device can't be set by `posix.time.clock_settime()`. Its tick count always increments, starting at 0 after power on. It is useful for timer/counters that generate events in relative time intervals, unaffected by changes to a wall clock.

4.12. /dev/dio

read: yes	write: yes	seek: yes	poll: [IN]/[OUT]/PRI/-	ioctl: no

Provides read and write access to the state of the digital I/O pins.

When read, a string of eight characters followed by a newline is returned, representing the state of the eight digital I/O pins in ascending order. Disabled pins are represented by a dash (-). Output pins are represented by ‘S’ if they are set/high, or by ‘R’ if they are reset/low. Input pins are either ‘1’ or ‘0’.

When writing the state, each of the eight pins must be represented in ascending order by one character. The write operation must end in a newline. Disabled and input pins must be set to a dash (-). Output pins can be set to ‘S’ or ‘1’ to drive them high, or to ‘R’ or ‘0’ to drive them low. Output pins can also be set to a dash (-) if they should keep their current state.

After each completed read or write operation, the file position must be rewinded to 0 before issuing the next operation.

The PRI event is signaled when the actual state of the input pins is different than it was during the last successful read operation. When the state was never read since opening the file, the behavior of the event is undefined.

4.13. /dev/dioc

read: yes	write: yes	seek: yes	poll: [IN]/[OUT]/-/-	ioctl: no

Provides read and write access to the configuration of the digital I/O pins.

When read, a string of eight characters followed by a newline is returned, representing the configuration of the eight digital I/O pins in ascending order. Disabled pins are represented by ‘d’. Output pins are represented by ‘o’. Input pins are represented by either ‘i’, ‘1’ or ‘0’ to indicate if they have no pull resistor, a pull-up or a pull-down respectively.

When writing the configuration, each of the eight pins must be represented in ascending order by one character. The write operation must end in a newline. The same mapping of characters applies as for reading the configuration (see above). In addition, a dash (-) can be used to indicate that the configuration of a pin should not be changed by the write operation.

After each completed read or write operation, the file position must be rewinded to 0 before issuing the next operation.

5. DOCUMENT HISTORY

Revision	Date	Changes
0	2017-10-06	First issue
1	2018-09-11	Added timer functions (ev.create_tim, evobj:settime, …) Added target package Added SPI package Fixed ev.waitany and ev.waitall return value description Renamed target.poweroff() Added example for target.systemoff() Added crypto.aes128() Added Memory Technology Device (MTD) functions Added target.mount() and target.umount() Added device file overview Added /dev/null, /dev/zero, /dev/ttyS0 and /dev/ttyAT Added /dev/ttyTIO0 Added /dev/mtd* Added /dev/tim* Added /dev/dio*

SUPPORT INQUIRIES

Link to **www.telit.com** and contact our technical support team for any questions related to technical issues.

www.telit.com

Telit Communications S.p.A.
Via Stazione di Prosecco, 5/B
I-34010 Sgonico (Trieste), Italy

Telit IoT Platforms LLC
5300 Broken Sound Blvd, Suite 150
Boca Raton, FL 33487, USA

Telit Wireless Solutions Inc.
3131 RDU Center Drive, Suite 135
Morrisville, NC 27560, USA

Telit Wireless Solutions Co., Ltd.
8th Fl., Shinyoung Securities Bld.
6, Gukjegeumyung-ro8-gil, Yeongdeungpo-gu
Seoul, 150-884, Korea

Telit Wireless Solutions Ltd.
10 Habarzel St.
Tel Aviv 69710, Israel

Telit Wireless Solutions
Technologia e Servicos Ltda
Avenida Paulista, 1776, Room 10.C
01310-921 São Paulo, Brazil