

Arch Rock IP/6LoWPAN Evaluation Software Distribution

1.0.0

Generated by Doxygen 1.5.5

Fri May 2 15:41:50 2008

Preliminary

Contents

1 Arch Rock IP/6LoWPAN Software Distribution (ASD)	1
1.1 Introduction	1
2 Interface Design Overview	5
2.1 API Types	6
2.2 Main() Entry Point	7
2.3 Error Handling	7
2.4 Concurrency Model	7
2.5 Interrupt Context	8
2.6 Hardware Abstraction Layer	10
2.7 Resource Descriptor Model	10
2.8 Memory Handling and Allocation	10
2.9 UDP Fragmentation Buffer and TCP Queue Buffer	11
2.10 Power Management	11
2.11 Watchdog and System Reboot	11
2.12 Route and Network Dynamics	12
2.13 Network Address Endianness	12
2.14 Remote Software Update	12
3 Function overview	13
4 ASD Tutorials	17
5 Data Structure Index	19

5.1	Data Structures	19
6	File Index	21
6.1	File List	21
7	Data Structure Documentation	23
7.1	event Struct Reference	23
7.2	in6_addr Struct Reference	25
7.3	itimerval Struct Reference	26
7.4	Ipstate Struct Reference	27
7.5	net_device Struct Reference	28
7.6	net_device_info Struct Reference	30
7.7	ping_cbargs Struct Reference	32
7.8	rtentry Struct Reference	34
7.9	sockaddr_in6 Struct Reference	35
7.10	tcp_event Struct Reference	36
7.11	timeval Struct Reference	37
7.12	udp_rcvfrom Struct Reference	38
8	File Documentation	39
8.1	include/errno.h File Reference	39
8.2	include/event.h File Reference	41
8.3	include/iwconfig.h File Reference	43
8.4	include/lowpan/lpstate.h File Reference	47
8.5	include/mainpage.h File Reference	48
8.6	include/net/icmp.h File Reference	49
8.7	include/net/route.h File Reference	51
8.8	include/netinet/in.h File Reference	54
8.9	include/netinet/inet.h File Reference	55
8.10	include/notifychange.h File Reference	58
8.11	include/platform/avr/platform.h File Reference	60
8.12	include/platform.h File Reference	61

8.13 include/sys/async.h File Reference	62
8.14 include/sys/flash.h File Reference	66
8.15 include/sys/socket.h File Reference	69
8.16 include/sys/svcs.h File Reference	77
8.17 include/sys/time.h File Reference	81
8.18 include/unistd.h File Reference	85

Preliminary

Preliminary

Chapter 1

Arch Rock IP/6LoWPAN Software Distribution (ASD)

1.1 Introduction

This distribution provides kernel library APIs for developers to create applications using Arch Rock embedded IPv6/6LoWPAN technology. Wherever applicable, the interfaces are designed to be as close to the well-known Unix socket interfaces as possible.

The kernel library goes beyond 6LoWPAN and supports some common set of protocols above IPv6 as well as some common embedded operating systems functionality that are useful for simplifying application development.

The kernel provides a single-thread [event](#) or callback driven concurrency model. It does not support dynamic memory allocation. Please refer to [Concurrency Model](#) and [Memory Handling and Allocation](#) sections for details.

The following shows the overview diagram of the kernel library, which includes the following major API components.

- Kernel Services and interrupts: reboot and power management control, or other kinds of network or processor notifications via [svcs.h](#), [async.h](#), and [platform.h](#).
- EEPROM Management: provide access to the internal/on-chip eeprom storage, which is shared with the kernel. This is platform dependent and is supported via [flash.h](#)
- Timers and Time Services: provide interval timer abstractions, get time of day services if synchronized with Arch Rock gateways via [time.h](#).
- Ping6: ability to initiate ICMP ping request via [icmp.h](#)

- TCP/IPv6: supports Transmission Control Protocol (TCP) over IPv6 via [socket.h](#).
- UDP/IPv6: supports User Datagram Protocol (UDP) over IPv6 via [socket.h](#).
- Route Table Management: ability to set static routes and access the routing table via [route.h](#).
- Wireless 15.4 Radio Configuration: abstract the 15.4 radio with a familiar network interface card abstraction via [iwconfig.h](#).

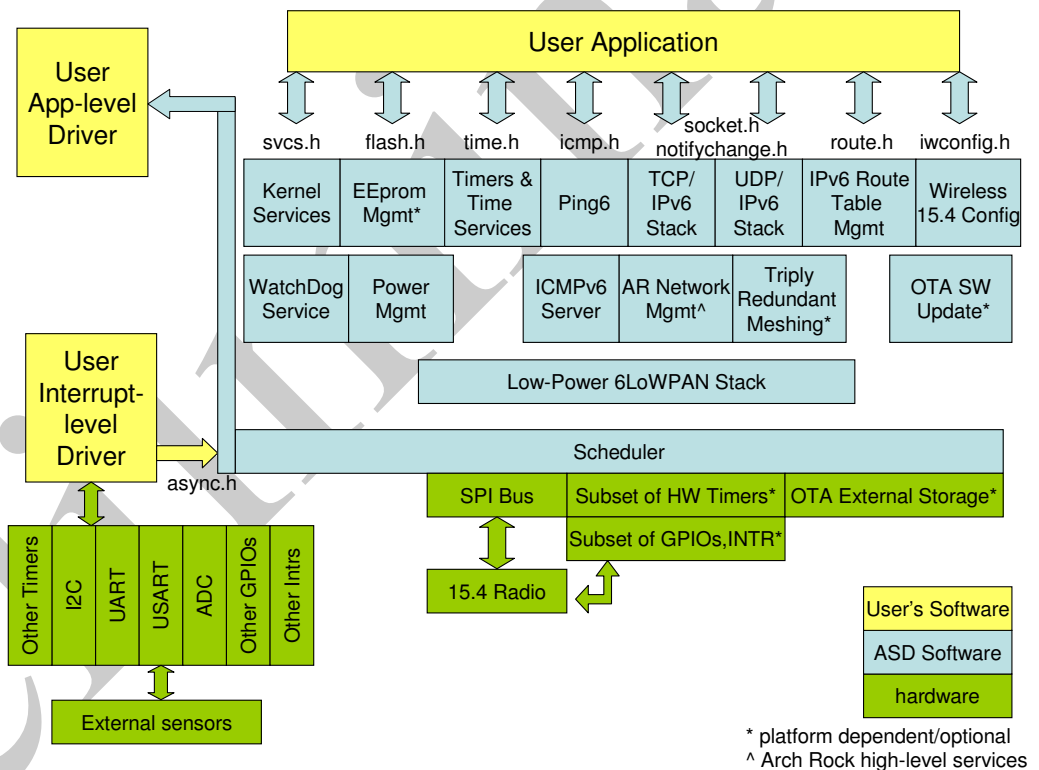


Figure 1.1: Overview Diagram

The kernel also supports a set of features not directly exposed through the APIs.

- Power Management: the kernel automatically manages the power of the processor and the 15.4 radio.

- ICMPv6 Server: respond to different ICMPv6 requests.
- AR Network Management: network-level management services with Arch Rock gateways.
- Triply Redundant Meshing: self-configure IPv6 meshing over 6LoWPAN.
- Over-the-Air Software Update: this depends on the platform such as the existence of an external flash to store programs.

For driver development, we try not to provide a hardware abstraction layer for the on-chip I/O peripherals. This gives developers maximum flexibility in configuring the I/Os for their desire purposes. They can either service the corresponding interrupts that they have setup to interface with the sensors or busy wait inside a callback handler from the kernel library.

A driver is usually written in two parts. One is within the interrupt context, where fast actions are required to be serviced. Heavy processing will continue in the application level context by scheduling a callback through the scheduler, as indicated in the diagram above. See [Interrupt Context](#) for details.

[Interface Design Overview](#)

[Function overview](#)

[ASD Tutorials](#)

Preliminary

Chapter 2

Interface Design Overview

Preliminary

2.1 API Types

This distribution does not support multi-threaded programming due to space constraints of the target platforms. Instead, asynchrony and concurrency are handled through callbacks. All of the APIs in this offering fall into one of the following categories.

- Function call with no callback.
- Function call with callback capability. This requires 3 additional arguments in the function signature:
 1. *a function pointer*: callback handler registration. If NULL is passed, the next two arguments are ignored. The call may fail return -1 or may succeed and return 0 without generating any notification.
 2. *a type specific pointer to hold callback arguments*: instead of providing callback arguments via function signature, data structures are used to hold the callback arguments. Caller allocates the structure, with the type defined by the particular function call, passes the pointer during the call, and the results will be stored in the structure upon the callback. If NULL is passed, the call may fail and return -1 or may succeed and return 0 for callback that takes no argument or callback that tolerates the caller to not take any callback arguments. Note that for the case of no argument callback, this automatically functions the same as a user context pointer described in the following next item.
 3. *a user context pointer*: this provides the convenience for users to maintain context along with continuation. The kernel merely passes the pointer along the continuation callback.
- Callback handler. All handlers from our APIs take the following function signature.

```
void(*) (event_t event, void * cbargs,
        void * context)
```

Parameters:

event passed by the kernel to specify the type of **event** and potential error information using **event_t**.

cbargs pointer to the return argument data structure passed to the kernel during call time

context pointer to the user context passed to the kernel during call time

Note:

event allows callback handler to inspect what function call the handler binds to originally. This is done by using the function enumeration name defined in **EventsEnum** and check it against **event.type**. Below is an example showing how a handler can tell the callback is originated from a timer function call. The subtype field in **event** is defined by the function call itself.

```
...
id = timer();
setitimer(id, &new_timeval, &old_timeval,
          app_handler, NULL, NULL);
...

app_handler(event_t event, void * cbargs, void * context)
{
    if (event.type == SETITIMER)
        ....
}
```

- Kernel and Interrupt Notifications. All notifications take the same function signature as the callback handler. A call must be made to bind a particular type of kernel or interrupt notification and to a function handler. This binding may recur instead of being just a one-time notification. The binding is removed when an explicit call (e.g. stop or unbind) is made to not receive this particular kernel notification.

2.2 Main() Entry Point

The first user code entry point from the kernel is a kernel notification with the name `__main()`.

```
void __main(event_t event, void * cbargs, void * context)
```

Since this shares the same signature as any callback handler in our APIs, one can reuse this same handler over and over again to create a continuation loop.

2.3 Error Handling

All function calls return 0 for success. All function calls fail by returning -1 and setting `errno` appropriately except for those that can be called within the interrupt context. These special calls are defined in [async.h](#) and they return the appropriate `errno` when error occurs.

Callback handler may entail call-specific error information in data structure specified in `cbargs`. Kernel notifications may entail error information by setting `event.error` to `event.errno` appropriately.

2.4 Concurrency Model

Our kernel library provides a single-threaded `event` driven concurrency model, with each `event` realized as a callback handler scheduled by our kernel's scheduler. The

scheduler ensures that callback handlers are atomic with respect to each other. That is, no callback can run until the current callback handler is completed.

Note:

This strictly requires that each callback handler must run into completion, which means writing a while(1) loop in the handler can hang the entire system.

However, interrupts can run and preempt any callback handlers. When interrupt handling is completed, the interrupted callback handler resumes. Race conditions can occur if any shared variables between the interrupt handler and the callback handler is not handled properly. One approach is to schedule a continuation callback handler in the interrupt handler and access the shared variable when the callback handler fires to maintain atomicity. Another approach is to directly control atomicity using atomic APIs defined in [async.h](#).

2.5 Interrupt Context

Our APIs cannot be called in the interrupt context except for those that schedule a continuation in the kernel context or those that control global interrupts for atomicity.

Our offering relies on timer and SPI interrupts to drive the network stack and kernel services. The design guideline is to handle fast process in interrupt context and schedule a continuation in the application context for more involved processing. This is exemplified by the following diagram.

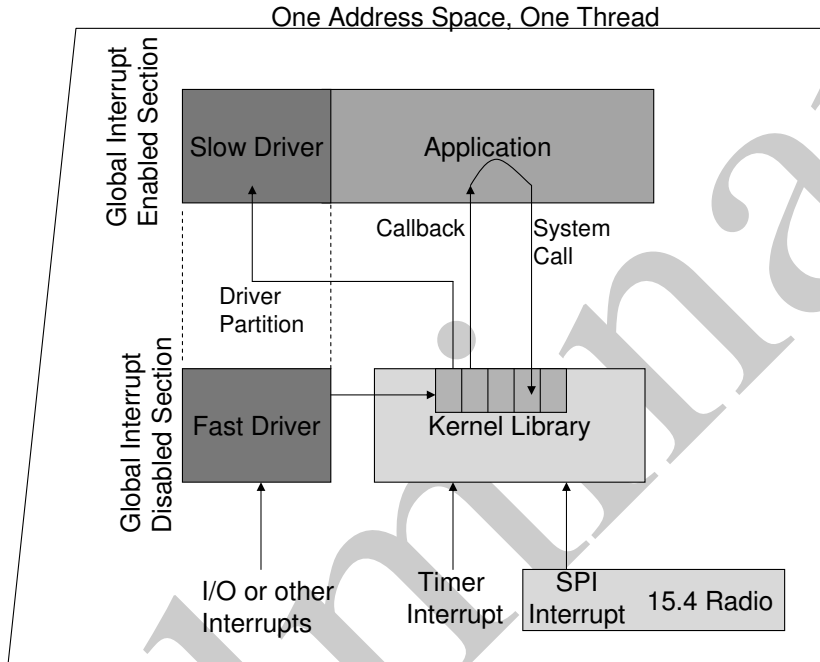


Figure 2.1: Overall Concurrency Diagram

A driver is usually written in two parts. One is within the interrupt context, where fast actions are required to be serviced. Heavy processing will continue in the application level context by scheduling a callback through the scheduler.

Busy waiting to handle I/O is possible within a callback handler as long as the busy waiting loop does not run "indefinitely". Like any event-driven systems, long busy waiting loop within a callback handler can take control of the system. To avoid the situation, one can utilize a timer to poll for an I/O [event](#) periodically to avoid waiting for I/O within an infinite loop.

The kernel owns the SPI bus, the necessary I/O pins, and interrupts to drive the 15.4 radio. It also uses a subset of hardware timers to support its time service. The kernel may also use a subset of other interrupt vectors, which driver developers may need to get shared access. This is a common case for some platforms in which the same

interrupt vector may be mapped to more than one kind of interrupt. These interrupts are all listed in *platform.h*

For these interrupts, the kernel will service interrupt handler if it is intended for the kernel. Otherwise, the kernel will ignore it and notify the application if there is an user interrupt handler binds to this interrupt. Refer to *async.h* to see how to bind an interrupt handler with the kernel. Note that once a user interrupt handler has called the bind function to an interrupt shared with the kernel, it will always get notified *whenever* the underlying interrupt fires, even if the firing is intended for the kernel. This is to simplify the kernel for small footprint rather than administrating filtering mechanism. Therefore, it is recommended that user defined interrupt handlers that share interrupts with the kernel should double check the interrupt flags to ensure the user is servicing the interrupt that he or she desires.

The list of interrupts used by or shared with the kernel is listed in *platform.h*. For other interrupts, user can connect to the interrupt vector directly.

2.6 Hardware Abstraction Layer

We do not offer any hardware abstraction layer on the processor except the internal on-chip non-volatile storage. Developers must create the relevant code themselves to access the peripherals and link it up to the kernel library.

2.7 Resource Descriptor Model

Resources that have multiple instances would require a file descriptor access model. These include socket, timer, and scheduling for a continuation in application context. When the number of instances for each type of resources has reached its limit, application must close some opened ones for reuse. Refer to *RELEASE_NOTES* to learn the limits.

2.8 Memory Handling and Allocation

The kernel does not provide dynamic memory APIs. All memory should be statically allocated.

Memory ownership of *cbargs* in callback handlers is returned back to the application upon the firing of the callback function. Kernel never owns the memory region pointed by *context*. It is purely intended for user context.

For callbacks that bind a resource descriptor with a callback handler, *cbargs* is owned by the kernel until the resource is closed explicitly by a call. Thus, application must

copy contents of the *cbargs* during each callback instance if it desires to retain the information.

Similarly, calls that explicitly require application buffers to operate require binding the buffer to a resource descriptor initially. The resulting buffer is released back to the application when the resource is closed explicitly.

2.9 UDP Fragmentation Buffer and TCP Queue Buffer

Our kernel library handles buffer queuing for both UDP and TCP communications and buffers are passed to the kernel through either *udpbind* or *tcpbind* calls.

For UDP, the buffer used is purely for fragmentation such that an application buffer is used to support a single application data unit (message) that is larger than the supported maximum transfer unit (MTU in [net_device_info.device](#)) in 15.4 radios. The buffer is used for the underlying mechanism to support link-layer fragmentation and reassembly. If no fragmentation is used, no buffer is required to pass down for UDP communications.

For TCP, the goal is to use an application buffer to queue up the byte streams for transmission and reception. Applications may want to adjust the buffer size, depending on the amount of backlog it expects or observes with its intended bandwidth requirement. Ideally, users should choose a large enough buffer so that they do not have to manage buffer back-pressure themselves. When back-pressure does occur, users may need to do some additional buffering of data above TCP and rely on *notifywrite* from the kernel to notify the user that more buffering is available for transmission.

2.10 Power Management

Power management of the radio and process is handled automatically by the kernel library. Application can override the policy and keep the CPU from sleeping. Duty-cycling powering of the radio can also be controlled separately. Application can be notified before processor goes to sleep each time if an handler is installed through kernel services in [svcs.h](#).

2.11 Watchdog and System Reboot

Application can reboot the system via a kernel service call or it can get a notification that a reboot is about to start when such reboot is administrated by remote Arch Rock management tools.

Applications can get notification for each watchdog [event](#) in the kernel to guard against its own invariant. The frequency of the watchdog [event](#) is controlled by the kernel and is

chosen to be 0.5 second. So, application only gets notified for each watchdog [event](#). If some invariants fail, application needs to explicitly call reboot through kernel services.

2.12 Route and Network Dynamics

Applications can get notifications on network dynamics such as route changes or that the node has joined or left a network. In addition, applications can add or remove IPv6 static routes as well as examining the entries in the routing table. A host only option is also available to disable a node from participating in routing traffic of others.

2.13 Network Address Endianness

APIs that have arguments with network address types, such as defining the destination IP address or the next hop address, expect big or network address endianness. If the endianness of the host is different, applications need to convert endianness using the provided APIs in [inet.h](#).

2.14 Remote Software Update

Depending on the platform, remote over-the-air software update can be supported in the future.

Chapter 3

Function overview

Preliminary

ICMP

ping() Perform the standard ICMP ping service to an unicast IP address or to a link local multicast address.

IPV6 ROUTING

num_rtrtries() Get the number of valid route entries in the routing table.

routecntl() Get and configure the routing table. If successful, the routes in the route table can be extracted or changed.

SOCKET PROGRAMMING

socket() Create an endpoint for communication and return a descriptor

send() Send a message on a socket in a connected state.

sendto() Send a message on a socket in a connectionless state.

udpbind() Give a UDP socket the local address, a user allocated buffer for message queuing, and a callback for reception notification.

tcpbind() Give a TCP socket the local address, a user allocated buffer for message queuing, and a callback handler for interacting with the TCP stack.

connect() Connect the socket referred to by the socket descriptor to an destination IP address.

accept() Accept an incoming connection on the socket referred to by the socket descriptor to an destination IP address.

inet_pton() Convert the character string src into a network address structure in the af address family, then copies the network address structure to dst.

inet_ntop() Convert a network address structure to the character string in the af address family.

ntohs() *htons()* *ntohl()* *htonl()* Convert byte order between network and host for short and long data types.

CONTEXT SWITCHING

continuation() Request a unique resource identifier that enables scheduling a continuation from the interrupt context to Arch Rock's application context. Note this call is safe to be invoked in the interrupt context.

sch_continuation() Schedule a continuation from the interrupt context to Arch Rock's application context. Note this call is safe to be invoked in the interrupt context.

continuation_close() Close a continuation resource identifier so that it may be reused. *close()* is not used because this supports call within the interrupt context.

INTERRUPT MANIPULATION/CRITICAL REGION PROTECTION

atomic_begin() Disable global interrupt for the start of a critical section.

atomic_end() Enable global interrupt for the end of a critical section.

async_irqbind() Bind a callback handler to a particular interrupt vector that are partially used by or shared with the kernel.

async_irqunbind() Unbind a previously installed callback handler to a particular interrupt vector that are partially used by or shared with the kernel.

TIME MANIPULATION

gettimeofday() Access time of day.

settimeofday() Set time of day.

timer() Request a unique resource identifier.

gettimer() Get value of an interval timer.

setitimer() Set the value of an interval timer.

FLASH MANIPULATION

fread() Read the on-chip flash

fsize() Get the total size of the internal non-volatile storage in bytes that are available for application to use.

fwrite() Write a buffer to the on-chip flash

SYSTEM SERVICES

close() Closes a resource identifier (e.g socket, timerid, or a resource id) so that it may be reused.

iwconfig() Configure the wireless link interface

notifywrite() Register a callback for when a socket identifier has enough buffer space to accept data for transmission.

svccntl() Invoke kernel services or listen for kernel notifications.

Preliminary

Chapter 4

ASD Tutorials

Preliminary

The tutorials are meant to demonstrate the ASD API by progressively more complex coding example, to get started, refer to projects/doc/doc_main.html

Preliminary

Chapter 5

Data Structure Index

5.1 Data Structures

Here are the data structures with brief descriptions:

event	23
in6_addr	25
itimerval	26
lpstate	27
net_device (IEEE 802.15.4 network link device abstraction)	28
net_device_info (IEEE 802.15.4 network link device information)	30
ping_cbargs	32
rtnetif	34
sockaddr_in6	35
tcp_event	36
timeval	37
udp_rcvfrom	38

Preliminary

Chapter 6

File Index

6.1 File List

Here is a list of all files with brief descriptions:

include/errno.h	39
include/event.h	41
include/iwconfig.h	43
include/mainpage.h	48
include/notifychange.h	58
include/platform.h	61
include/unistd.h	85
include/lowpan/lpstate.h	47
include/net/icmp.h	49
include/net/route.h	51
include/netinet/in.h	54
include/netinet/inet.h	55
include/platform/avr/platform.h	60
include/sys/async.h	62
include/sys/flash.h	66
include/sys/socket.h	69
include/sys/svcs.h	77
include/sys/time.h	81

Preliminary

Chapter 7

Data Structure Documentation

7.1 event Struct Reference

```
#include <event.h>
```

Data Fields

- [uint8_t type](#)
- [int16_t subtype](#)
- [int16_t flags](#)
- [int8_t error](#)

7.1.1 Field Documentation

7.1.1.1 uint8_t event::type

See [EventsEnum](#).

7.1.1.2 int16_t event::subtype

More type information. This is call specific.

7.1.1.3 int16_t event::flags

Notifications may entail flag information. This is call specific.

7.1.1.4 int8_t event::error

Notifications may entail error information defined in [ErrnoEnum](#).

The documentation for this struct was generated from the following file:

- [include/event.h](#)

7.2 in6_addr Struct Reference

```
#include <in.h>
```

Data Fields

- union {
 - uint8_t u6_addr8 [16]
 - uint16_t u6_addr16 [8]
 - uint32_t u6_addr32 [4]

```
} in6_u
```

7.2.1 Detailed Description

IPv6 address data structure.

7.2.2 Field Documentation

7.2.2.1 uint8_t in6_addr::u6_addr8[16]

IP Address as an array of bytes

7.2.2.2 uint16_t in6_addr::u6_addr16[8]

IP Address as an array of shorts

7.2.2.3 uint32_t in6_addr::u6_addr32[4]

IP Address as an array of 32-bit integers

7.2.2.4 union { ... } in6_addr::in6_u

The documentation for this struct was generated from the following file:

- include/netinet/[in.h](#)

7.3 itimerval Struct Reference

```
#include <time.h>
```

Data Fields

- struct [timeval it_interval](#)
- struct [timeval it_value](#)

7.3.1 Detailed Description

Interval timer value.

7.3.2 Field Documentation

7.3.2.1 struct timeval itimerval::it_interval [read]

Value to put into 'it_value' when the timer expires.

7.3.2.2 struct timeval itimerval::it_value [read]

Time to the next timer expiration.

The documentation for this struct was generated from the following file:

- [include/sys/time.h](#)

7.4 lpstate Struct Reference

```
#include <lpstate.h>
```

Data Fields

- uint16_t [listen_period](#)
- uint16_t [chirp_period](#)

7.4.1 Detailed Description

Arch Rock Low Power Protocol Parameter Settings.

7.4.2 Field Documentation

7.4.2.1 uint16_t lpstate::listen_period

Specify the radio wake up period in milliseconds Recommended default is 64ms.

7.4.2.2 uint16_t lpstate::chirp_period

Specify the radio chirp period in milliseconds. Recommended default is to keep this the same as listen period.

The documentation for this struct was generated from the following file:

- [include/lowpan/lpstate.h](#)

7.5 net_device Struct Reference

IEEE 802.15.4 network link device abstraction.

```
#include <iwconfig.h>
```

Data Fields

- [uint8_t name](#)
- [uint16_t mtu](#)
- [uint8_t dev_addr](#) [MAX_ADDR_LEN]
- [lpstate_t lpstate](#)
- [uint8_t channel](#)
- [uint16_t panid](#)
- [uint8_t key](#) [KEYSIZE]
- [int16_t tx_pwr](#)

7.5.1 Detailed Description

IEEE 802.15.4 network link device abstraction.

7.5.2 Field Documentation

7.5.2.1 uint8_t net_device::name

Name of the link interface.

Note:

The default 15.4 radio interface name is *LPAN0*. This is fixed by the kernel library and cannot be changed by developers.

7.5.2.2 uint16_t net_device::mtu

MTU size of link interface. This cannot be changed.

7.5.2.3 uint8_t net_device::dev_addr[MAX_ADDR_LEN]

Link (MAC) address of the device.

7.5.2.4 lpstate_t net_device::lpstate

Low power configurations.

7.5.2.5 uint8_t net_device::channel

15.4 channel

7.5.2.6 uint16_t net_device::panid

15.4 PAN ID.

7.5.2.7 uint8_t net_device::key[KEYSIZE]

Security key.

7.5.2.8 int16_t net_device::tx_pwr

15.4 radio transmission power in discrete units of dBm. (e.g. 0dBm) The actual values are provided through proper enumerations.

Note:

The supported range is manufacturer dependent.

The documentation for this struct was generated from the following file:

- [include/iwconfig.h](#)

7.6 net_device_info Struct Reference

IEEE 802.15.4 network link device information.

```
#include <iwconfig.h>
```

Data Fields

- [net_device_t device](#)
- [in6_addr_t v6_gaddr](#)
- [in6_addr_t v6_laddr](#)
- [in6_addr_t v6_prefix](#)
- [uint16_t packets_sent](#)
- [uint16_t packets_rcv](#)
- [uint16_t flags](#)

7.6.1 Detailed Description

IEEE 802.15.4 network link device information.

7.6.2 Field Documentation

7.6.2.1 net_device_t net_device_info::device

Device data structure

7.6.2.2 in6_addr_t net_device_info::v6_gaddr

Global IPv6 Address of the interface.

7.6.2.3 in6_addr_t net_device_info::v6_laddr

Link Local IPv6 Address of the interface.

7.6.2.4 in6_addr_t net_device_info::v6_prefix

IPv6 Prefix of the interface.

7.6.2.5 uint16_t net_device_info::packets_sent

Number of 15.4 packets sent.

7.6.2.6 uint16_t net_device_info::packets_recv

Number of 15.4 packets received.

7.6.2.7 uint16_t net_device_info::flags

Flags reflecting current status of the interface.

The documentation for this struct was generated from the following file:

- [include/iwconfig.h](#)

7.7 ping_cbargs Struct Reference

```
#include <icmp.h>
```

Data Fields

- [sockaddr_in6_t src](#)
- [int8_t rssi](#)
- [uint16_t delay](#)
- [uint8_t len](#)
- [const char * ping_reply](#)
- [int8_t error](#)
- [uint8_t done](#)

7.7.1 Field Documentation

7.7.1.1 `sockaddr_in6_t ping_cbargs::src`

source address of the ping reply

7.7.1.2 `int8_t ping_cbargs::rssi`

Received signal strength if *src* is 1-hop away.

7.7.1.3 `uint16_t ping_cbargs::delay`

Ping delay in milliseconds.

7.7.1.4 `uint8_t ping_cbargs::len`

Length of the ping reply in bytes.

7.7.1.5 `const char* ping_cbargs::ping_reply`

String buffer that contains a user readable text message to show the result of the Reply. For example, "12 bytes from fe80::17:3b00:0:15 rssi=-35 dBm delay=135ms" or "Request timed out".

7.7.1.6 int8_t ping_cbargs::error

Ping errors: SUCCESS, ETIMEOUT, ENETUNREACH. See [ErrnoEnum](#).

7.7.1.7 uint8_t ping_cbargs::done

True if this is the last signal for ping

The documentation for this struct was generated from the following file:

- [include/net/icmp.h](#)

7.8 rentry Struct Reference

```
#include <route.h>
```

Data Fields

- [in6_addr_t dst](#)
- [in6_addr_t nxthop](#)
- [uint8_t hops](#)
- [uint16_t cost](#)

7.8.1 Field Documentation

7.8.1.1 in6_addr_t rentry::dst

For this destination.

7.8.1.2 in6_addr_t rentry::nxthop

Next hop address for the above destination. This has to be a IPv6 link local address.

7.8.1.3 uint8_t rentry::hops

7.8.1.4 uint16_t rentry::cost

The documentation for this struct was generated from the following file:

- [include/net/route.h](#)

7.9 sockaddr_in6 Struct Reference

```
#include <socket.h>
```

Data Fields

- [uint16_t sin6_port](#)
- [in6_addr_t sin6_addr](#)
- [uint8_t sin6_flowinfo](#)
- [uint8_t sin6_flags](#)

7.9.1 Detailed Description

IPv6 Socket address and port

7.9.2 Field Documentation

7.9.2.1 [uint16_t sockaddr_in6::sin6_port](#)

Transport layer port

7.9.2.2 [in6_addr_t sockaddr_in6::sin6_addr](#)

IPv6 address

7.9.2.3 [uint8_t sockaddr_in6::sin6_flowinfo](#)

IPv6 flow information. Reserved.

7.9.2.4 [uint8_t sockaddr_in6::sin6_flags](#)

Reserved.

The documentation for this struct was generated from the following file:

- [include/sys/socket.h](#)

7.10 tcp_event Struct Reference

```
#include <socket.h>
```

Data Fields

- [int16_t socket](#)
- [int8_t type](#)
- [void * recvbuf](#)
- [int16_t recvlen](#)
- [sockaddr_in6_t addr](#)

7.10.1 Field Documentation

7.10.1.1 int16_t tcp_event::socket

Returns the same socket id during call time

7.10.1.2 int8_t tcp_event::type

Reflect the state of the underlying TCP session.

7.10.1.3 void* tcp_event::recvbuf

7.10.1.4 int16_t tcp_event::recvlen

7.10.1.5 sockaddr_in6_t tcp_event::addr

The socket address of the message's sender.

The documentation for this struct was generated from the following file:

- [include/sys/socket.h](#)

7.11 timeval Struct Reference

```
#include <time.h>
```

Data Fields

- uint32_t [tv_sec](#)
- uint32_t [tv_usec](#)

7.11.1 Detailed Description

A time value that is accurate to the nearest microsecond but also has a range of years.

7.11.2 Field Documentation

7.11.2.1 uint32_t timeval::tv_sec

Seconds.

7.11.2.2 uint32_t timeval::tv_usec

Microseconds.

The documentation for this struct was generated from the following file:

- [include/sys/time.h](#)

7.12 udp_recvfrom Struct Reference

```
#include <socket.h>
```

Data Fields

- `int16_t sockid`
- `void * payload`
- `uint8_t len`
- `sockaddr_in6_t * from`

7.12.1 Field Documentation

7.12.1.1 `int16_t udp_recvfrom::sockid`

returns the same socket id during call time

7.12.1.2 `void* udp_recvfrom::payload`

points to the received message stored in the queue.

7.12.1.3 `uint8_t udp_recvfrom::len`

length of the message in number of bytes.

7.12.1.4 `sockaddr_in6_t* udp_recvfrom::from`

the socket address of the message's sender.

The documentation for this struct was generated from the following file:

- `include/sys/socket.h`

Chapter 8

File Documentation

8.1 include/errno.h File Reference

```
#include <platform.h>
```

Defines

- #define `_ERRNO_H` 1

Enumerations

- enum `ErrnoEnum` {
 `FAIL` = -1, `SUCCESS` = 0, `ESIZE` = 2, `ECANCEL` = 3,
 `EOFF` = 4, `EBUSY` = 5, `EINVAL` = 6, `ERETRY` = 7,
 `ERESERVE` = 8, `ENETUNREACH` = 9, `ENOTTIMESYNC` = 10, `ETIMEOUT`
 = 11,
 `ENFILE` = 12, `ENOTCONN` = 13 }

Error number enumerations.

Variables

- `int16_t` `errno`

8.1.1 Define Documentation

8.1.1.1 #define _ERRNO_H 1

8.1.2 Enumeration Type Documentation

8.1.2.1 enum ErrnoEnum

Error number enumerations.

Enumerator:

FAIL Generic indication of a failed completion of a call or operation.

SUCCESS Generic indication of a successful completion of a call or operation.

ESIZE Parameter passed in was too big.

ECANCEL Operation cancelled by a call.

EOFF Subsystem is not active

EBUSY The underlying system is busy; retry later

EINVAL An invalid parameter was passed.

ERETRY A rare and transient failure: can retry

ERESERVE Reservation required before usage

ENETUNREACH Destination is unreachable

ENOTTIMESYNC Global time is not synchronized

ETIMEOUT Operation timed out

ENFILE Descriptor limit has been reached.

ENOTCONN Socket is not connected.

8.1.3 Variable Documentation

8.1.3.1 int16_t errno

8.2 include/event.h File Reference

```
#include <platform.h>
```

Data Structures

- struct [event](#)

Defines

- #define [_EVENT_H 1](#)

Typedefs

- typedef struct [event](#) [event_t](#)

Enumerations

- enum [EventsEnum](#) {
 [MAIN](#) = 1, [IWCONFIG](#), [UDPBIND](#), [TCPBIND](#),
 [PING](#), [SCH_CONTINUATION](#), [SVCCNTL](#), [SETITIMER](#),
 [NOTIFYWRITE](#), [ASYNC_IRQBIND](#) }

8.2.1 Define Documentation

8.2.1.1 #define [_EVENT_H 1](#)

8.2.2 Typedef Documentation

8.2.2.1 typedef struct [event](#) [event_t](#)

8.2.3 Enumeration Type Documentation

8.2.3.1 enum [EventsEnum](#)

Enumerations for type in [event_t](#) for callbacks. The name of the enumerations exactly matches the original calls that register the callbacks.

Enumerator:

MAIN

IWCONFIG
UDPBIND
TCPBIND
PING
SCH_CONTINUATION
SVCCNTL
SETTIMER
NOTIFYWRITE
ASYNC_IRQBIND

8.3 include/iwconfig.h File Reference

```
#include <event.h>
#include <netinet/in.h>
#include <lowpan/lpstate.h>
```

Data Structures

- struct [net_device](#)
IEEE 802.15.4 network link device abstraction.
- struct [net_device_info](#)
IEEE 802.15.4 network link device information.

Defines

- #define [IWCONFIG_H](#) 1

Typedefs

- typedef struct [net_device](#) [net_device_t](#)
- typedef struct [net_device_info](#) [net_device_info_t](#)

Enumerations

- enum [iwconfig_options](#) {
[O_AUTHEN](#) = 0x1, [O_ENCRYPT](#) = 0x2, [O_SET_MAC_ADDRESS](#) = 0x4,
[O_SET_KEY](#) = 0x8,
[O_SET_TX_POWER](#) = 0x12, [O_NOT_DEFAULT_LP](#) = 0x14, [O_NO_LP](#) =
0x16 }
- enum [net_device_status_flags](#) { [IFF_DOWN](#) = 0x1, [IFF_UP](#) = 0x2, [IFF_-](#)
[LOOPBACK](#) = 0x4, [IFF_MULTICAST](#) = 0x8 }
- enum { [MAX_ADDR_LEN](#) = 8, [KEYSIZE](#) = 16 }
- enum [InterfaceEnum](#) { [LPAN0](#) = 1 }
- enum [IwConfigCmdsEnum](#) { [UP](#) = 1, [DOWN](#), [CONFIG](#), [STATUS](#) }

Functions

- `int16_t iwconfig` (`uint8_t name`, `uint16_t cmd`, `uint16_t options`, `const net_device_t *device_settings`, `void(*iwconfig_handler)(event_t event, void *cbargs, void *context)`, `net_device_info_t *dev_info`, `void *context`)

8.3.1 Define Documentation

8.3.1.1 `#define IWCONFIG_H 1`

8.3.2 Typedef Documentation

8.3.2.1 `typedef struct net_device_info net_device_info_t`

8.3.2.2 `typedef struct net_device net_device_t`

8.3.3 Enumeration Type Documentation

8.3.3.1 anonymous enum

Storage sizes for device data structure abstraction.

Enumerator:

MAX_ADDR_LEN 64 bit MAC Address

KEYSIZE 128 bit security key size

8.3.3.2 `enum InterfaceEnum`

Supported network interfaces.

Enumerator:

LPAN0

8.3.3.3 `enum iwconfig_options`

Common configuration options for IEEE 15.4 radio. Multiple options can coexist as flags.

Enumerator:

O_AUTHEN Enable authentication mode if supported

O_ENCRYPT Enable encryption mode if supported
O_SET_MAC_ADDRESS Set 64 bit MAC 15.4 address
O_SET_KEY Set security key if supported
O_SET_TX_POWER Set the radio's transmission power
O_NOT_DEFAULT_LP Use non-default low power settings.
O_NO_LP Set the radio to not use low power protocol.

8.3.3.4 enum IwConfigCmdsEnum

iwconfig commands

Enumerator:

UP
DOWN
CONFIG
STATUS

8.3.3.5 enum net_device_status_flags

Flags reflecting the current status of the 15.4 radio.

Enumerator:

IFF_DOWN Link is down.
IFF_UP Link is up and running.
IFF_LOOPBACK Link loopback is supported
IFF_MULTICAST Link local IPv6 multicast is supported

8.3.4 Function Documentation

8.3.4.1 `int16_t iwconfig (uint8_t name, uint16_t cmd, uint16_t options, const net_device_t * device_settings, void (*)(event_t event, void *cbargs, void *context) iwconfig_handler, net_device_info_t * dev_info, void * context)`

A system call to control and configure the wireless link interface. A continuation call-back will be fired if *iwconfig_handler* is not NULL.

Parameters:

- ← *name* Interface name (e.g. The default 6LowPAN interface is "Ipan0")
- ← *cmd* Support [UP, DOWN, CONFIG, STATUS]. See commands [IwConfigCmdsEnum](#)
 - *UP* power up interface
 - *DOWN* power down interface
 - *CONFIG* configure the radio based on *options* and *device_settings*
 - *STATUS* fills *dev_info* with current interface status
- ← *options* See [iwconfig_options](#).
- ← *device_settings* Application specifies radio settings. Required for UP and CONFIG *cmd*.
- ← *iwconfig_handler* A continuation callback for the completion of the system call. Passing NULL here will fail the call.
- *dev_info* the results of the calls for the continuation callback are stored here
NULL cannot be passed.
- ← *context* application context

Returns:

0 if succeed or -1 if fail and errno is set appropriately.

Errors:

EBUSY if the underlying system is busy and cannot service the call at this time.

EINVAL if invalid parameters are passed.

8.3.5 Callback Descriptions

8.3.5.1 iwconfig_handler

Parameters:

- ← *event* Type is IWCONFIG in [EventsEnum](#)
- ← *cbargs* is *dev_info* in [iwconfig\(\)](#)
- ← *context* is *context* in [iwconfig\(\)](#)

Returns:

None.

8.4 include/lowpan/lpstate.h File Reference

Data Structures

- struct [lpstate](#)

Defines

- #define [_LPSTATE_H 1](#)

Typedefs

- typedef struct [lpstate](#) [lpstate_t](#)

8.4.1 Define Documentation

8.4.1.1 #define [_LPSTATE_H 1](#)

8.4.2 Typedef Documentation

8.4.2.1 typedef struct [lpstate](#) [lpstate_t](#)

8.5 include/mainpage.h File Reference

Preliminary

8.6 include/net/icmp.h File Reference

```
#include <event.h>
#include <sys/socket.h>
```

Data Structures

- struct [ping_cbargs](#)

Defines

- #define [_ICMP_H](#) 1

Typedefs

- typedef struct [ping_cbargs](#) [ping_cbargs_t](#)

Functions

- int16_t [ping](#) (const char *dst, void(*ping_handler)([event_t event](#), void *cbargs, void *context), [ping_cbargs_t](#) *ping_result, void *context)

8.6.1 Define Documentation

8.6.1.1 #define [_ICMP_H](#) 1

8.6.2 Typedef Documentation

8.6.2.1 typedef struct [ping_cbargs](#) [ping_cbargs_t](#)

8.6.3 Function Documentation

8.6.3.1 int16_t [ping](#) (const char * *dst*, void(*)([event_t event](#), void **cbargs*, void **context*) *ping_handler*, [ping_cbargs_t](#) * *ping_result*, void * *context*)

A system call to perform the standard ICMP ping service to an unicast IP address or to a link local multicast address.

The ping service is either completed by getting a ping response from the destination or a timeout notification. In the case of link local multicast with multiple responses, the *ping_handler()* will be invoked for each ping respond. A timeout notification will

always be generated to signal the completion of a link local multicast ping. The *done* field in *ping_cbargs_t* signals the completion of the ping and thus expires the binding of the callback handler with the ping call.

Parameters:

- ← *dst* IPv6 address of the desired destination(s) to ping
- ← *ping_handler* continuation callback handler. If NULL is passed, call will fail.
- *ping_result* results of the calls for the continuation callback are stored here. if NULL is passed, call will fail
- ← *context* application context

Returns:

0 if succeed and -1 if fail and errno is set appropriately.

Errors:

- EBUSY if the underlying system is busy
- EIO if underlying system is off. (e.g. link interface is off)
- EINVAL if invalid parameters are passed

8.6.4 Callback Descriptions

8.6.4.1 ping_handler

Parameters:

- ← *event* Type is PING in [EventsEnum](#)
- ← *cbargs* is *ping_result* in [ping\(\)](#)
- ← *context* is *context* in [ping\(\)](#)

Returns:

None.

8.7 include/net/route.h File Reference

```
#include <sys/socket.h>
```

Data Structures

- struct [rtenry](#)

Defines

- #define [_ROUTE_H 1](#)

Typedefs

- typedef struct [rtenry](#) [rtenry_t](#)

Enumerations

- enum [RouteCmdsEnum](#) { [ADDRROUTE](#) = 30, [DELETEROUTE](#), [READROUTE](#) }

Functions

- [int16_t routectl](#) ([uint16_t cmd](#), [int16_t rtindex](#), [rtenry_t *rtenry](#))
- [int16_t num_rtenries](#) ()
A system call to get the number of valid route entries in the routing table.

8.7.1 Define Documentation

8.7.1.1 #define [_ROUTE_H 1](#)

8.7.2 Typedef Documentation

8.7.2.1 typedef struct [rtenry](#) [rtenry_t](#)

8.7.3 Enumeration Type Documentation

8.7.3.1 enum [RouteCmdsEnum](#)

Commands for [routectl\(\)](#).

Enumerator:

ADDROUTE
DELETEROUTE
READROUTE

8.7.4 Function Documentation**8.7.4.1 int16_t num_rtrentries ()**

A system call to get the number of valid route entries in the routing table.

Precondition:

Do not call this inside the interrupt handler context.

Postcondition:

If successful, return the number of routes.

Returns:

number of valid route entries in the routing table

8.7.4.2 int16_t routecntl (uint16_t cmd, int16_t rtindex, rtrentry_t * rtrentry)

A system call to get and configure the routing table. If successful, the routes in the route table can be extracted or changed.

Parameters:

← *cmd* See commands [RouteCmdsEnum](#)

- *ADDROUTE* to add a static route entry
- *DELETEROUTE* to delete a static route entry
- *READROUTE* read the route entry, indexed by *rtindex*, into *rtrentry*

← *rtindex* index to the route table

↔ *rtrentry* contains the entry of the routing table to be set to or extracted from the table

Returns:

0 if succeed and -1 if fail and errno is set appropriately.

Errors:

EINVAL if invalid parameters are passed

ESIZE if index is larger than table size

Preliminary

8.8 include/netinet/in.h File Reference

```
#include <platform.h>
```

Data Structures

- struct [in6_addr](#)

Defines

- #define [_NETINET_IN_H](#) 1
- #define [s6_addr](#) in6_u.u6_addr8
- #define [s6_addr16](#) in6_u.u6_addr16
- #define [s6_addr32](#) in6_u.u6_addr32

Typedefs

- typedef struct [in6_addr](#) [in6_addr_t](#)

8.8.1 Define Documentation

8.8.1.1 #define [_NETINET_IN_H](#) 1

8.8.1.2 #define [s6_addr](#) in6_u.u6_addr8

8.8.1.3 #define [s6_addr16](#) in6_u.u6_addr16

8.8.1.4 #define [s6_addr32](#) in6_u.u6_addr32

8.8.2 Typedef Documentation

8.8.2.1 typedef struct [in6_addr](#) [in6_addr_t](#)

8.9 include/netinet/inet.h File Reference

```
#include <netinet/in.h>
```

Defines

- #define `_INET_IN_H` 1

Functions

- `int16_t inet_pton` (`int16_t af`, `const char *src`, `in6_addr_t *dst`)
- `const char * inet_ntop` (`int16_t af`, `const in6_addr_t *src`, `char *dst`, `uint8_t cnt`)
- `uint16_t ntohs` (`uint16_t netshort`)
- `uint16_t htons` (`uint16_t hostshort`)
- `uint32_t ntohl` (`uint32_t netlong`)
- `uint32_t htonl` (`uint32_t hostlong`)

8.9.1 Define Documentation

8.9.1.1 #define `_INET_IN_H` 1

8.9.2 Function Documentation

8.9.2.1 `uint32_t htonl` (`uint32_t hostlong`)

Convert a host byte order (little-endian) long, *hostlong*, by returning a network byte order (big-endian) long.

8.9.2.2 `uint16_t htons` (`uint16_t hostshort`)

Convert a host byte order (little-endian) short, *hostshort*, by returning a network byte order (big-endian) short.

8.9.2.3 `const char* inet_ntop` (`int16_t af`, `const in6_addr_t * src`, `char * dst`, `uint8_t cnt`)

This system call converts a network address structure defined in *src* to the character string *dst* using *af* address family.

Precondition:

dst is a buffer large enough for the family address

Postcondition:

dst contains the string corresponding to the address in *src*

Parameters:

- ← *af* AF address family. See [AFEnums](#).
- ← *src* contains an IP address string
- *dst* string buffer
- ← *cnt* number of bytes in the buffer

Returns:

- dst if succeed or NULL if fail and errno is set appropriately
- EINVAL if invalid parameters are passed

8.9.2.4 int16_t inet_pton (int16_t af, const char * src, in6_addr_t * dst)

This system call converts the character string *src* into a network address structure in the *af* address family, then copies the network address structure to *dst*.

Precondition:

src is a null terminated string.

Postcondition:

dst contains the correct network address corresponds to the *src*

Parameters:

- ← *af* AF address family. See [AFEnums](#).
- ← *src* contains an IP address string
- *dst* copy into this network data structure

Returns:

- 0 if succeed or -1 if fail and errno is set appropriately
- EINVAL if invalid parameters are passed

8.9.2.5 uint32_t ntohl (uint32_t netlong)

Convert a network byte order (big-endian) long, *netlong*, by returning a host byte order (little-endian) long.

8.9.2.6 uint16_t ntohs (uint16_t *netshort*)

Convert a network byte order (big-endian) short, *netshort*, by returning a host byte order (little-endian) short.

8.10 include/notifychange.h File Reference

```
#include <event.h>
```

Defines

- #define `_NOTIFYCHANGE_H` 1

Functions

- `int16_t notifywrite` (`int16_t resid`, `void(*notify_handler)(event_t event, void *cbargs, void *context)`, `void *resource`, `void *context`)

8.10.1 Define Documentation

8.10.1.1 #define `_NOTIFYCHANGE_H` 1

8.10.2 Function Documentation

8.10.2.1 `int16_t notifywrite` (`int16_t resid`, `void(*) (event_t event, void *cbargs, void *context) notify_handler`, `void * resource`, `void * context`)

Callback when the underlying socket specified by the socket identifier has more buffer again and is ready to accept data for transmission.

Parameters:

- ← *resid* a valid socket id.
- ← *notify_handler* the continuation callback handler that will be fired when the underlying socket can accept more data. If NULL is passed here, the call will fail.
- ← *resource* no requirement here.
- ← *context* user context

Returns:

0 if succeed or -1 if fail and errno is set appropriately.

Errors:

EBUSY if the underlying system cannot accept this request.

EINVAL if invalid parameters are passed

8.10.3 Callback Descriptions

8.10.3.1 notify_handler

Parameters:

- ← *event* Type is NOTIFYWRITE in [EventsEnum](#).
- ← *cbargs* return *resource* in [notifywrite\(\)](#).
- ← *context* return *context* in [notifywrite\(\)](#).

Returns:

None.

8.11 include/platform/avr/platform.h File Reference

```
#include <inttypes.h>
#include <stdlib.h>
```

Defines

- #define `__PLATFORM_H` 1

Enumerations

- enum `InterruptTypeEnum` { `WATCHDOGTIMER`, `CPUSLEEP`, `ASYNC_COUNT` }

8.11.1 Define Documentation

8.11.1.1 #define `__PLATFORM_H` 1

8.11.2 Enumeration Type Documentation

8.11.2.1 enum `InterruptTypeEnum`

Enumerator:

WATCHDOGTIMER CPU WatchDog Notification. Bind callback handler to kernel when started. Handler will be fired whenever Kernel's watchdog timer fires.

CPUSLEEP CPU Sleep Notification. Bind callback handler to kernel when started. Handler will be fired before CPU goes to sleep.

ASYNC_COUNT

8.12 include/platform.h File Reference

```
#include <platform/avr/platform.h>
```

8.13 include/sys/async.h File Reference

```
#include <event.h>
```

Defines

- #define `_ASYNC_H` 1

Functions

- void `atomic_begin` ()
- void `atomic_end` ()
- int16_t `continuation` ()
- int16_t `sch_continuation` (int16_t resid, void(*app_handler)(event_t event, void *cbargs, void *context), void *resource, void *context)
- int16_t `async_irqbind` (int16_t intr_enum, void(*async_io_handler)(event_t event, void *cbargs, void *context), void *resource, void *context)
- int16_t `async_irqunbind` (int16_t intr_enum)
- int16_t `continuation_close` (int16_t id)

8.13.1 Define Documentation

8.13.1.1 #define _ASYNC_H 1

8.13.2 Function Documentation

8.13.2.1 int16_t `async_irqbind` (int16_t *intr_enum*, void(*)*(event_t event, void *cbargs, void *context) async_io_handler*, void * *resource*, void * *context*)

Bind a callback handler to a particular interrupt vector that are partially used by or shared with the kernel. Kernel services that requires interrupt context notification should use this to bind with notification handler. Note that this API is platform dependent and the handler will be fired within interrupt context. Once bind has called, subsequent call of this function will replace the old handler with a new handler.

Parameters:

- ← *intr_enum* platform dependent interrupt enumeration number. Refer to [InterruptTypesEnum](#) for details.
- ← *async_handler* the continuation callback handler that will be fired in interrupt context. If NULL is passed here, the call will fail.

- ← *resource* pass NULL, will be ignored
- ← *context* pass NULL, will be ignored

Returns:

0 if succeed or EINVAL if invalid. Note errno is not used because this is in the interrupt context.

8.13.3 Callback Descriptions

8.13.3.1 `async_handler`

Parameters:

- ← *event* type is ASYNC_IO in [EventsEnum](#). subtype defines which interrupt is firing using [InterruptTypesEnum](#) defined in platform specific header files.
- ← *cbargs* return NULL
- ← *context* return NULL

Returns:

None.

8.13.3.2 `int16_t async_irqunbind (int16_t intr_enum)`

Unbind a previously installed callback handler to a particular interrupt vector that are partially used by or shared with the kernel. Note that this API is platform dependent and the handler will be fired within interrupt context. Once unbind has called, subsequent interrupts firing will not be notified.

Parameters:

- ← *intr_enum* platform dependent interrupt enumeration number. Refer to [InterruptTypesEnum](#) for details.

Returns:

0 if succeed or EINVAL if invalid. Note errno is not used because this is in the interrupt context.

8.13.3.3 `void atomic_begin ()`

Disable global interrupt for the start of a critical section.

8.13.3.4 void atomic_end ()

Enable global interrupt for the end of a critical section.

8.13.3.5 int16_t continuation ()

Request a unique resource identifier that enables scheduling a continuation from the interrupt context to Arch Rock's application context. Note this call is safe to be invoked in the interrupt context.

Returns:

a non-zero resource identifier if succeed or ENFILE all unique resource identifier has been used up. Note that errno is not used since this is in the interrupt context.

8.13.3.6 int16_t continuation_close (int16_t id)

A system call that closes a continuation identifier so that it may be reused. This is safe to be called within the interrupt context.

Returns:

0 on success, EINVAL for bad identifier, and EBUSY for waiting for an action to be executed before resource identifier can be freed.

8.13.3.7 int16_t sch_continuation (int16_t resid, void(*) (event_t event, void *cbargs, void *context) app_handler, void *resource, void *context)

To schedule a continuation from the interrupt context to Arch Rock's application context. Note this call is safe to be invoked in the interrupt context.

Parameters:

- ← *resid* a valid resource id return by [continuation\(\)](#).
- ← *app_handler* the continuation callback handler that will be fired in application context. If NULL is passed here, the call will fail.
- ← *resource* no requirement here.
- ← *context* user context

Returns:

0 if succeed or EBUSY if the resource with *resid* is busy or EINVAL if invalid parameters are passed. Note that errno is not used because this is in the interrupt context.

8.13.4 Callback Descriptions

8.13.4.1 app_handler

Parameters:

- ← *event* Type is SCH_CONTINUATION in [EventsEnum](#).
- ← *cbargs* return *resource* in [sch_continuation\(\)](#).
- ← *context* return *context* in [sch_continuation\(\)](#).

Returns:

None.

8.13.5 Code Example

```
#include <sys/async.h>

int16_t g_resid;

void __TIMER1_OVF_vect() {
    // interrupt context code
    g_resid = continuation();
    sch_continuation(g_resid, user_handler, NULL, NULL);
}

void user_handler(event_t event, void * cbaargs, void * context) {
    // ... user context action here
}
```

8.14 include/sys/flash.h File Reference

Defines

- #define `_FLASH_H 1`

Functions

- `int16_t flwrite` (const void *frombuf, uint16_t size, uint16_t toaddr)
- `int16_t fread` (uint16_t fromaddr, void *tobuf, uint16_t size)
- `uint16_t flsize` ()

8.14.1 Define Documentation

8.14.1.1 #define `_FLASH_H 1`

8.14.2 Function Documentation

8.14.2.1 `int16_t fread (uint16_t fromaddr, void * tobuf, uint16_t size)`

A system call to provide random read access of the on-chip flash as a non-volatile storage.

Precondition:

Do not call this inside the interrupt handler context. Caller must maintain its own flash address management to avoid reading unwanted data. 0 is the start address of the flash abstraction.

Postcondition:

If successful, *size* number of bytes of *fromaddr* on the flash is written to *tobuf*.

Parameters:

- ← *fromaddr* address of the flash where data is read from
- *tobuf* address of a the buffer where content will be copied to
- ← *size* size of *tobuf* in bytes

Returns:

0 if succeed or -1 if fail and errno is set appropriately.

Errors:

EINVAL if invalid parameters are passed

ESIZE if size is too large

8.14.2.2 uint16_t flsize ()

A function call that returns the total size of the internal non-volatile storage in bytes that are available for application to use.

Precondition:

None.

Postcondition:

returns the total size of the internal non-volatile storage in bytes that are available to applications.

Returns:

size of the internal non-volatile storage in bytes

8.14.2.3 int16_t flwrite (const void * *frombuf*, uint16_t *size*, uint16_t *toaddr*)

A system call to provide random write access to the on-chip flash as a non-volatile storage.

Precondition:

Do not call this inside the interrupt handler context. Caller must maintain its own flash address management to avoid overwriting previous data. 0 is the start address of the flash abstraction.

Postcondition:

If successful, *size* number of bytes of *frombuf* is written into the flash starting at *toaddr*.

Parameters:

- ← *frombuf* pointer to the buffer to be written to flash
- ← *size* size of the buffer in bytes
- ← *toaddr* start address on the flash where content will be copied to

Returns:

0 if succeed or -1 if fail and errno is set appropriately.

Errors:

EINVAL if invalid parameters are passed

ESIZE if size is too large

Preliminary

8.15 include/sys/socket.h File Reference

```
#include <netinet/in.h>
```

```
#include <event.h>
```

Data Structures

- struct [sockaddr_in6](#)
- struct [udp_recvfrom](#)
- struct [tcp_event](#)

Defines

- #define [_SOCKET_H](#) 1
- #define [s6_addr](#) in6_u.u6_addr8
- #define [s6_addr16](#) in6_u.u6_addr16
- #define [s6_addr32](#) in6_u.u6_addr32

Typedefs

- typedef struct [sockaddr_in6](#) [sockaddr_in6_t](#)
- typedef struct [udp_recvfrom](#) [udp_recvfrom_t](#)
- typedef struct [tcp_event](#) [tcp_event_t](#)

Enumerations

- enum [SockTypeEnums](#) { [SOCK_STREAM](#) = 1, [SOCK_DGRAM](#), [SOCK_DGRAM_FRAG](#) }
- enum [AFEnums](#) { [AF_INET](#) = 10, [AF_INET6](#) }
- enum [TcpStatesEnum](#) { [ACCEPT](#) = 1, [CONNECTED](#), [RECV](#), [CLOSED](#) }

Functions

- [int16_t socket](#) ([int16_t](#) domain, [int16_t](#) type, [int16_t](#) protocol)
- [int16_t send](#) ([int16_t](#) s, const void *buf, [int16_t](#) len, [int16_t](#) flags)
- [int16_t sendto](#) ([int16_t](#) s, const void *buf, [int16_t](#) len, [int16_t](#) flags, const [sockaddr_in6_t](#) *to)
- [int16_t udpbind](#) ([int16_t](#) s, const [sockaddr_in6_t](#) *my_addr, void *buf, [int16_t](#) buflen, void(*recvfrom)([event_t](#) event, void *cbargs, void *context), [udp_recvfrom_t](#) *recvfrom_cbargs, void *context)

- `int16_t tcpbind` (`int16_t s`, `const sockaddr_in6_t *my_addr`, `void *buf`, `int16_t buflen`, `void(*tcp_handler)(event_t event, void *cbargs, void *context)`, `tcp_event_t *tcp_event`, `void *context`)
- `int16_t connect` (`int16_t s`, `const sockaddr_in6_t *to`)
- `int16_t accept` (`int16_t s`, `const sockaddr_in6_t *to`)

8.15.1 Define Documentation

8.15.1.1 `#define _SOCKET_H 1`

8.15.1.2 `#define s6_addr in6_u.u6_addr8`

8.15.1.3 `#define s6_addr16 in6_u.u6_addr16`

8.15.1.4 `#define s6_addr32 in6_u.u6_addr32`

8.15.2 Typedef Documentation

8.15.2.1 `typedef struct sockaddr_in6 sockaddr_in6_t`

8.15.2.2 `typedef struct tcp_event tcp_event_t`

8.15.2.3 `typedef struct udp_recvfrom udp_recvfrom_t`

8.15.3 Enumeration Type Documentation

8.15.3.1 `enum AFEnums`

AF_INET Protocol Family

Enumerator:

AF_INET IPv4 Internet protocols

AF_INET6 IPv6 Internet protocols

8.15.3.2 `enum SocketTypeEnums`

Socket types.

Enumerator:

SOCK_STREAM Provides sequenced, reliable, two-way, connection-based byte streams.

SOCK_DGRAM Supports datagrams (connectionless, unreliable messages of a fixed maximum length bounded by link layer)

SOCK_DGRAM_FRAG Supports datagrams (connectionless, unreliable messages of a fixed maximum length bounded supported by link layer fragmentation.) This is not currently supported.

8.15.3.3 enum TcpStatesEnum

Enumerator:

ACCEPT Incoming TCP connection. `addr` in `tcp_event_t` contains the IP address of the peer initiating the connection.

CONNECTED TCP connection established.

RECV TCP connection has incoming data.

CLOSED TCP connection is closed.

8.15.4 Function Documentation

8.15.4.1 int16_t accept (int16_t s, const sockaddr_in6_t * to)

A system call that accept an incoming connection on the socket referred to by the socket descriptor to an destination IP address. Only the type SOCK_STREAM is supported for this call.

Precondition:

This call should be invoked after `tcpbind()` has notified that the socket has changed to the ACCEPT state.

Postcondition:

The system will try to service the incoming connection to the destination.

Parameters:

← *s* socket descriptor

← *to* destination IP address

Returns:

0 if the system agrees to make a connection. On error, -1 is returned, and `errno` is set appropriately.

Errors:

EINVAL invalid argument passed

EBUSY underlying socket is busy (already connected).

8.15.4.2 `int16_t connect (int16_t s, const sockaddr_in6_t * to)`

A system call that connects the socket referred to by the socket descriptor to an destination IP address. Only the type SOCK_STREAM is supported for this call.

This is a non-blocking connect call. Actual establishment of the connection or failure will be notified later.

Precondition:

This call must be invoked after `tcpbind()`.

Postcondition:

The system tries to initiate a connection to the destination.

Parameters:

← *s* socket descriptor

← *to* destination IP address

Returns:

0 if the system agrees to make a connection. On error, -1 is returned, and `errno` is set appropriately.

Errors:

EINVAL invalid argument passed

EBUSY underlying socket is busy (already connected).

8.15.4.3 `int16_t send (int16_t s, const void * buf, int16_t len, int16_t flags)`

A system call to send a message on a socket to another socket. The socket must be in a connected state (so that the intended recipient is known).

When completed, the system has copied *buf* up to *len* number of bytes into its internal queue for in order transmission of the byte stream.

Parameters:

← *s* socket descriptor of the sending socket.

- ← *buf* buffer that contains the message. The system will copy the message into the user provided buffer queue established in *tcpbind*.
- ← *len* length of the message in bytes.
- ← *flags* reserved for future use.

Returns:

0 is returned on success. On error, -1 is returned, and *errno* is set appropriately.

Errors:

EINVAL invalid argument passed. ESIZE no buffer space to accommodate message
ENOTCONN the socket is not connected

8.15.4.4 int16_t sendto (int16_t s, const void * buf, int16_t len, int16_t flags, const sockaddr_in6_t * to)

A system call to send a message on a socket to another socket. The socket must be in a connectionless state with the intended recipient specified as an argument.

When completed, the system has copied *buf* up to *len* number of bytes into its internal queue for transmission.

Parameters:

- ← *s* socket descriptor of the sending socket.
- ← *buf* buffer that contains the message. The system will copy the message into the user provided buffer queue established in *udpbind*.
- ← *len* length of the message in bytes.
- ← *flags* reserved for future use.
- ← *to* recipient address in socket address structure

Returns:

On error, -1 is returned, and *errno* is set appropriately.

Errors:

EINVAL invalid argument passed. ESIZE no buffer space to accommodate message at this time.

8.15.4.5 `int16_t` socket (`int16_t domain`, `int16_t type`, `int16_t protocol`)

A system call to create an endpoint for communication and return a descriptor if the total limit has not been reached.

Parameters:

- ← *domain* specifies a communication domain or the protocol family which will be used for communication. See [AFEnums](#).
- ← *type* specifies the communication semantics. See [SockTypeEnums](#).
- ← *protocol* Not supported.

Returns:

On success, a socket descriptor for the new socket is returned. On error, -1 is returned, and *errno* is set appropriately.

Errors:

EINVAL unknown protocol, or protocol family not available.

ENFILE the system limit on the total number of open sockets with this protocol has been reached.

8.15.4.6 `int16_t` tcpbind (`int16_t s`, `const sockaddr_in6_t * my_addr`, `void * buf`, `int16_t buflen`, `void(*) (event_t event, void *cbargs, void *context) tcp_handler`, `tcp_event_t * tcp_event`, `void * context`)

A system call that gives a TCP socket the local address, a user allocated buffer for message queuing, and a callback handler for interacting with the TCP stack.

The system binds the socket *s* to *my_addr*, the corresponding callback handlers, and the user-provided heap memory for transmission queuing. Listening on the address is automatically started. These bindings will be cleared when the socket is closed.

Parameters:

- ← *s* socket descriptor of the socket
- ← *my_addr* local address to bind to the socket
- ← *buf* user allocated heap memory to be used for buffering
- ← *buflen* length of *buf* in bytes.
- ← *tcp_handler* a callback handler to interact with the different states of a TCP session. If NULL is passed, call will fail.

→ *tcp_event* argument for the *tcp_handler* notification. if NULL is passed, call will fail.

← *context* user context

Returns:

On success, zero is returned. On error, -1 is returned and *errno* is set appropriately.

Errors:

EINVAL invalid argument passed.

EFAULT bad address.

8.15.5 Callback Descriptions

8.15.5.1 tcpbind

Parameters:

← *event* Type is TCPBIND in *EventsEnum*.

← *cbargs* is *tcp_event* in *tcpbind()*.

← *context* is *context* in *tcpbind()*.

Returns:

None.

8.15.5.2 int16_t udpbind (int16_t s, const sockaddr_in6_t * my_addr, void * buf, int16_t buflen, void(*) (event_t event, void *cbargs, void *context) recvfrom, udp_recvfrom_t * recvfrom_cbargs, void * context)

A system call that gives a UDP socket the local address, a user allocated buffer for message queuing, and a callback for reception notification.

When completed, the system binds socket *s* to *my_addr*, *recvfrom* handler for packet reception notification and listening on the address is started automatically. These bindings will be cleared when the socket is closed.

Parameters:

← *s* socket descriptor of the socket

← *my_addr* local address to bind to the socket

- ← *buf* user allocated heap memory to be used for buffering. (only for SOCK_DGRAM_FRAG)
- ← *buflen* length of *buf* in bytes.(only for SOCK_DGRAM_FRAG)
- ← *recvfrom* a callback handler that signals a UDP packet reception destined to *my_addr*. If NULL is passed, call will fail.
- *recvfrom_cbargs* storing the recvfrom arguments for the callback. If NULL, only notification will be generated.
- ← *context* user context

Returns:

On success, zero is returned. On error, -1 is returned and *errno* is set appropriately.

Errors:

EINVAL invalid argument passed.

EFAULT bad address.

Note:

recvfrom_cbargs needs to be copied if user wants to retain its information across callbacks.

8.15.6 Callback Descriptions

8.15.6.1 udpbind

Parameters:

- ← *event* Type is UDPBIND in [EventsEnum](#).
- ← *cbargs* is *recvfrom_cbargs* in [udpbind\(\)](#).
- ← *context* is *context* in [udpbind\(\)](#).

Returns:

None.

8.16 include/sys/svcs.h File Reference

```
#include <platform.h>
#include <event.h>
```

Defines

- #define `_SVCS_H` 1

Enumerations

- enum `SystemSvcsEnum` {
 `REBOOT_CNTL` = 1, `CPUPOWER_CNTL`, `ROUTESVC_CNTL`,
 `SWUPDATE_CNTL`,
 `CNTL_COUNT`, `REBOOT_NOTIFY`, `ROUTESVC_NOTIFY`, `SWUPDATE_`
 `NOTIFY`,
 `NOTIFY_COUNT` }
- enum `SvcCmdsEnum` { `START` = 10, `STOP` }
- enum `RouterSvcCmdsEnum` { `HOST` = 20, `ROUTER` }
- enum `RouteFlagsEnum` { `F_NETWORK_JOINED` = 1, `F_LEFT_NETWORK` }
- enum `SwUpdateCmdsEnum` { `DOWNLOAD_UPDATE` = 50, `PROBE_`
 `UPDATE`, `REPROGRAM` }
- enum `RebootFlagsEnum` { `F_REBOOT_BY_GW` = 0x1, `F_REPROGRAM_`
 `BY_GW` }

Functions

- `int16_t svccntl` (`int16_t` svc_enum, `uint16_t` cmd, `void(*svc_handler)(event_t`
 `event`, `void *cbargs`, `void *context`), `void *resource`, `void *context`)

8.16.1 Define Documentation

8.16.1.1 #define `_SVCS_H` 1

8.16.2 Enumeration Type Documentation

8.16.2.1 enum `RebootFlagsEnum`

Reboot Service flags in `event_t` on the callback notification.

Enumerator:

F_REBOOT_BY_GW
F_REPROGRAM_BY_GW

8.16.2.2 enum RouteFlagsEnum**Enumerator:**

F_NETWORK_JOINED Node has joined a 6LoWPAN Mesh Network.
F_LEFT_NETWORK Node has is not connected to a 6LoWPAN Mesh Network

8.16.2.3 enum RouterSvcCmdsEnum

Router service commands

Enumerator:

HOST
ROUTER

8.16.2.4 enum SvcCmdsEnum

Kernel service commands

Enumerator:

START
STOP

8.16.2.5 enum SwUpdateCmdsEnum

Software Update command Enum

Enumerator:

DOWNLOAD_UPDATE
PROBE_UPDATE
REPROGRAM

8.16.2.6 enum SystemSvcsEnum

Kernel services.

Enumerator:

REBOOT_CNTL Reboot Control. Execute directly. No callback is required. Supported commands: START and STOP.

CPUPOWER_CNTL CPU Sleep Control. Execute directly. No callback is required. Supported commands: START and STOP.

ROUTESVC_CNTL Route Control to configure if the node should be a router or just a host which does not route traffic from other nodes. Execute directly. No callback is required. Supported commands: HOST and ROUTER.

SWUPDATE_CNTL Software Update control. Not supported. Potential commands: PROBE_UPDATE, DOWNLOAD_UPDATE, and REPROGRAM.

CNTL_COUNT

Note:

REBOOT_NOTIFY The following notifications will use the *subtype* field in *event_t*.

Reboot Notification. Bind callback handler to kernel when started. Handler will be fired before node goes to reboot. Stopping the service removes the handler binding. Supported commands: START and STOP.

ROUTESVC_NOTIFY Route Changes Notification. Bind callback handler to kernel when started. Handler will be fire whenever route changes. The nature of the change is stored in *flags* field in *event_t* with values defined in [Route-FlagsEnum](#). Stopping the service removes the handler binding. Supported commands: START and STOP.

SWUPDATE_NOTIFY Software Update service. Not supported. Supported commands: PROBE_UPDATE, DOWNLOAD_UPDATE, REPROGRAM,

NOTIFY_COUNT

8.16.3 Function Documentation

8.16.3.1 `int16_t svccntl(int16_t svc_enum, uint16_t cmd, void(*)(event_t event, void *cbargs, void *context) svc_handler, void *resource, void *context)`

Invoke kernel services or listen for kernel notifications.

Parameters:

← *svc_enum* kernel service enumeration name. See [SystemSvcsEnum](#).

- ← *cmd* commands to invoke the service
- ← *svc_handler* the continuation callback handler for notification. If NULL is passed here, the call will fail unless *cmd* is for control and does not require notification.
- ← *resource* no requirement here.
- ← *context* user context

Returns:

0 if succeed or -1 if fail and errno is set appropriately.

Errors:

EBUSY if the underlying system is busy

EINVAL if invalid parameters are passed

8.16.4 Callback Descriptions

8.16.4.1 *svc_handler*

Parameters:

- ← *event* type is SVCCNTL in [EventsEnum](#). subtype is set to *cmd*. flags field may contain extra information, depending on nature of the *cmd*.
- ← *cbargs* return *resource* in *svncntl()*
- ← *context* return *context* in *svncntl()*

Returns:

None.

8.17 include/sys/time.h File Reference

```
#include <event.h>
```

Data Structures

- struct [timeval](#)
- struct [itimerval](#)

Defines

- #define [_TIME_H](#) 1

Typedefs

- typedef struct [timeval](#) [timeval_t](#)
- typedef struct [itimerval](#) [itimerval_t](#)

Functions

- int16_t [gettimeofday](#) (struct [timeval](#) *tv)
- int16_t [settimeofday](#) (const struct [timeval](#) *tv)
- int16_t [timer](#) ()
- int16_t [gettimer](#) (int16_t timerid, [itimerval_t](#) *value)
- int16_t [setitimer](#) (int16_t timerid, const [itimerval_t](#) *value, [itimerval_t](#) *old_value, void(*timer_handler)([event_t](#) event, void *cbargs, void *context), void *resource, void *context)

8.17.1 Define Documentation

8.17.1.1 #define _TIME_H 1

8.17.2 Typedef Documentation

8.17.2.1 typedef struct itimerval itimerval_t

8.17.2.2 typedef struct timeval timeval_t

8.17.3 Function Documentation

8.17.3.1 int16_t gettimeofday (struct timeval * tv)

A system call to access time of day.

Precondition:

Do not call this inside the interrupt handler context.

Postcondition:

If successful, *tv* contains the system's time of day.

Parameters:

→ *tv* system fills the time value structure with system's time of the day.

Returns:

0 if succeed or -1 if fail and errno is set appropriately.

Errors:

ENOTTIMESYNC suggests the returned time value is not synchronized.

8.17.3.2 int16_t gettimer (int16_t timerid, itimerval_t * value)

Get value of an interval timer.

Parameters:

← *timerid* a valid timer id return by [timer\(\)](#)

→ *value* to get the value of the timer

Returns:

0 if succeed or -1 if fail and errno is set appropriately.

Errors:

EINVAL if invalid parameters are passed

8.17.3.3 int16_t setitimer (int16_t *timerid*, const itimerval_t * *value*, itimerval_t * *old_value*, void(*)(*event_t* event, void **cbargs*, void **context*) *timer_handler*, void * *resource*, void * *context*)

Set the value of an interval timer. Setting the value to zero means stopping the current timer.

Parameters:

- ← *timerid* a valid timer id return by `timer()`.
- ← *value* new interval timer value
- *old_value* if old timer interval is not zero, it will copied into *old_value*. Pass NULL if don't care.
- ← *timer_handler* the continuation callback handler that will be fired when *value* becomes zero. If NULL is passed here, the call will fail.
- ← *resource* no requirement here.
- ← *context* user context

Returns:

0 if succeed or -1 if fail and `errno` is set appropriately.

Errors:

EBUSY if the resource with *timerid* is busy

EINVAL if invalid parameters are passed

8.17.4 Callback Descriptions**8.17.4.1 timer_handler****Parameters:**

- ← *event* Type is SETTIMER in `EventsEnum`.
- ← *cbargs* return *resource* in `setitimer()`.
- ← *context* return *context* in `setitimer()`.

Returns:

None.

8.17.4.2 int16_t settimeofday (const struct timeval * tv)

A system call to set system's time of day.

Precondition:

Do not call this inside the interrupt handler context.

Postcondition:

If successful, system's time of day will be set to that specified in *tv*.

Parameters:

← *tv* the time value structure that the system will take as time of day.

Returns:

0 if succeed or -1 if fail and errno is set appropriately.

Errors:

EINVAL invalid argument

8.17.4.3 int16_t timer ()

Request a unique resource identifier that enables scheduling timers.

Returns:

a non-zero resource identifier if succeed or -1 if fail and errno is set appropriately.

Errors:

ENFILE all unique resource identifier has been used up

8.18 include/unistd.h File Reference

Defines

- `#define _UNISTD_H 1`

Functions

- `int16_t close (int16_t id)`

8.18.1 Define Documentation

8.18.1.1 `#define _UNISTD_H 1`

8.18.2 Function Documentation

8.18.2.1 `int16_t close (int16_t id)`

A system call that closes a resource identifier (e.g socket, timerid, or a resource id) so that it may be reused.

Returns:

0 on success. On error, -1 is returned and `errno` is set appropriately.

Errors:

`EINVAL` bad identifier

`EBUSY` resource may require closing sessions before freeing resource identifier.

Index

`_ASYNC_H`
 [async.h, 62](#)

`_ERRNO_H`
 [errno.h, 40](#)

`_EVENT_H`
 [event.h, 41](#)

`_FLASH_H`
 [flash.h, 66](#)

`_ICMP_H`
 [icmp.h, 49](#)

`_INET_IN_H`
 [inet.h, 55](#)

`_IWCONFIG_H`
 [iwconfig.h, 44](#)

`_LPSTATE_H`
 [lpstate.h, 47](#)

`_NETINET_IN_H`
 [in.h, 54](#)

`_NOTIFYCHANGE_H`
 [notifychange.h, 58](#)

`_ROUTE_H`
 [route.h, 51](#)

`_SOCKET_H`
 [socket.h, 70](#)

`_SVCS_H`
 [svcs.h, 77](#)

`_TIME_H`
 [time.h, 82](#)

`_UNISTD_H`
 [unistd.h, 85](#)

`__PLATFORM_H`
 [platform/avr/platform.h, 60](#)

`ACCEPT`
 [socket.h, 71](#)

`accept`
 [socket.h, 71](#)

`addr`
 [tcp_event, 36](#)

`ADDRROUTE`
 [route.h, 52](#)

`AF_INET`
 [socket.h, 70](#)

`AF_INET6`
 [socket.h, 70](#)

`AFEnums`
 [socket.h, 70](#)

`async.h`
 [_ASYNC_H, 62](#)
 [async_irqbind, 62](#)
 [async_irqunbind, 63](#)
 [atomic_begin, 63](#)
 [atomic_end, 63](#)
 [continuation, 64](#)
 [continuation_close, 64](#)
 [sch_continuation, 64](#)

`ASYNC_COUNT`
 [platform/avr/platform.h, 60](#)

`ASYNC_IRQBIND`
 [event.h, 42](#)

`async_irqbind`
 [async.h, 62](#)

`async_irqunbind`
 [async.h, 63](#)

`atomic_begin`
 [async.h, 63](#)

`atomic_end`
 [async.h, 63](#)

`channel`
 [net_device, 29](#)

`chirp_period`
 [lpstate, 27](#)

`close`

unistd.h, 85

CLOSED
 socket.h, 71

CNTL_COUNT
 svcs.h, 79

CONFIG
 iwconfig.h, 45

connect
 socket.h, 72

CONNECTED
 socket.h, 71

continuation
 async.h, 64

continuation_close
 async.h, 64

cost
 rtenry, 34

CPUPOWER_CNTL
 svcs.h, 79

CPUSLEEP
 platform/avr/platform.h, 60

delay
 ping_cbargs, 32

DELETEROUTE
 route.h, 52

dev_addr
 net_device, 28

device
 net_device_info, 30

done
 ping_cbargs, 33

DOWN
 iwconfig.h, 45

DOWNLOAD_UPDATE
 svcs.h, 78

dst
 rtenry, 34

EBUSY
 errno.h, 40

ECANCEL
 errno.h, 40

EINVAL
 errno.h, 40

ENETUNREACH
 errno.h, 40

ENFILE
 errno.h, 40

ENOTCONN
 errno.h, 40

ENOTTIMESYNC
 errno.h, 40

EOFF
 errno.h, 40

ERESERVE
 errno.h, 40

ERETRY
 errno.h, 40

errno
 errno.h, 40

errno.h
 _ERRNO_H, 40
 EBUSY, 40
 ECANCEL, 40
 EINVAL, 40
 ENETUNREACH, 40
 ENFILE, 40
 ENOTCONN, 40
 ENOTTIMESYNC, 40
 EOFF, 40
 ERESERVE, 40
 ERETRY, 40
 errno, 40
 ErrnoEnum, 40
 ESIZE, 40
 ETIMEOUT, 40
 FAIL, 40
 SUCCESS, 40

ErrnoEnum
 errno.h, 40

error
 event, 23
 ping_cbargs, 32

ESIZE
 errno.h, 40

ETIMEOUT
 errno.h, 40

event, 23
 error, 23
 flags, 23
 subtype, 23

- type, 23
- event.h
 - _EVENT_H, 41
 - ASYNC_IRQBIND, 42
 - event_t, 41
 - EventsEnum, 41
 - IWCONFIG, 41
 - MAIN, 41
 - NOTIFYWRITE, 42
 - PING, 42
 - SCH_CONTINUATION, 42
 - SETTIMER, 42
 - SVCCNTL, 42
 - TCPBIND, 42
 - UDPBIND, 42
- event_t
 - event.h, 41
- EventsEnum
 - event.h, 41
- F_LEFT_NETWORK
 - svcs.h, 78
- F_NETWORK_JOINED
 - svcs.h, 78
- F_REBOOT_BY_GW
 - svcs.h, 78
- F_REPROGRAM_BY_GW
 - svcs.h, 78
- FAIL
 - errno.h, 40
- flags
 - event, 23
 - net_device_info, 31
- flash.h
 - _FLASH_H, 66
 - fread, 66
 - flsize, 67
 - flwrite, 67
- fread
 - flash.h, 66
- flsize
 - flash.h, 67
- flwrite
 - flash.h, 67
- from
 - udp_rcvfrom, 38
- gettimeofday
 - time.h, 82
- gettimer
 - time.h, 82
- hops
 - rtenry, 34
- HOST
 - svcs.h, 78
- htonl
 - inet.h, 55
- htons
 - inet.h, 55
- icmp.h
 - _ICMP_H, 49
 - ping, 49
 - ping_cbargs_t, 49
- IFF_DOWN
 - iwconfig.h, 45
- IFF_LOOPBACK
 - iwconfig.h, 45
- IFF_MULTICAST
 - iwconfig.h, 45
- IFF_UP
 - iwconfig.h, 45
- in.h
 - _NETINET_IN_H, 54
 - in6_addr_t, 54
 - s6_addr, 54
 - s6_addr16, 54
 - s6_addr32, 54
- in6_addr, 25
 - in6_u, 25
 - u6_addr16, 25
 - u6_addr32, 25
 - u6_addr8, 25
- in6_addr_t
 - in.h, 54
- in6_u
 - in6_addr, 25
- include/errno.h, 39
- include/event.h, 41
- include/iwconfig.h, 43
- include/lowpan/lpstate.h, 47
- include/mainpage.h, 48

include/net/icmp.h, 49
 include/net/route.h, 51
 include/netinet/in.h, 54
 include/netinet/inet.h, 55
 include/notifychange.h, 58
 include/platform.h, 61
 include/platform/avr/platform.h, 60
 include/sys/async.h, 62
 include/sys/flash.h, 66
 include/sys/socket.h, 69
 include/sys/svcs.h, 77
 include/sys/time.h, 81
 include/unistd.h, 85
 inet.h
 _INET_IN_H, 55
 htonl, 55
 htons, 55
 inet_ntop, 55
 inet_pton, 56
 ntohl, 56
 ntohs, 56
 inet_ntop
 inet.h, 55
 inet_pton
 inet.h, 56
 InterfaceEnum
 iwconfig.h, 44
 InterruptTypesEnum
 platform/avr/platform.h, 60
 it_interval
 itimerval, 26
 it_value
 itimerval, 26
 itimerval, 26
 it_interval, 26
 it_value, 26
 itimerval_t
 time.h, 82
 IWCONFIG
 event.h, 41
 iwconfig
 iwconfig.h, 45
 iwconfig.h
 _IWCONFIG_H, 44
 CONFIG, 45
 DOWN, 45
 IFF_DOWN, 45
 IFF_LOOPBACK, 45
 IFF_MULTICAST, 45
 IFF_UP, 45
 InterfaceEnum, 44
 iwconfig, 45
 iwconfig_options, 44
 IwConfigCmdsEnum, 45
 KEYSIZE, 44
 LPAN0, 44
 MAX_ADDR_LEN, 44
 net_device_info_t, 44
 net_device_status_flags, 45
 net_device_t, 44
 O_AUTHEN, 44
 O_ENCRYPT, 44
 O_NO_LP, 45
 O_NOT_DEFAULT_LP, 45
 O_SET_KEY, 45
 O_SET_MAC_ADDRESS, 45
 O_SET_TX_POWER, 45
 STATUS, 45
 UP, 45
 iwconfig_options
 iwconfig.h, 44
 IwConfigCmdsEnum
 iwconfig.h, 45
 key
 net_device, 29
 KEYSIZE
 iwconfig.h, 44

 len
 ping_cbargs, 32
 udp_recvfrom, 38
 listen_period
 lpstate, 27
 LPAN0
 iwconfig.h, 44
 lpstate, 27
 chirp_period, 27
 listen_period, 27
 net_device, 28
 lpstate.h
 _LPSTATE_H, 47

- lpstate_t, 47
- lpstate_t
 - lpstate.h, 47
- MAIN
 - event.h, 41
- MAX_ADDR_LEN
 - iwconfig.h, 44
- mtu
 - net_device, 28
- name
 - net_device, 28
- net_device, 28
 - channel, 29
 - dev_addr, 28
 - key, 29
 - lpstate, 28
 - mtu, 28
 - name, 28
 - panid, 29
 - tx_pwr, 29
- net_device_info, 30
 - device, 30
 - flags, 31
 - packets_recv, 30
 - packets_sent, 30
 - v6_gaddr, 30
 - v6_laddr, 30
 - v6_prefix, 30
- net_device_info_t
 - iwconfig.h, 44
- net_device_status_flags
 - iwconfig.h, 45
- net_device_t
 - iwconfig.h, 44
- NOTIFY_COUNT
 - svcs.h, 79
- notifychange.h
 - _NOTIFYCHANGE_H, 58
 - notifywrite, 58
- NOTIFYWRITE
 - event.h, 42
- notifywrite
 - notifychange.h, 58
- ntohl
 - inet.h, 56
- ntohs
 - inet.h, 56
- num_rtries
 - route.h, 52
- nxtHop
 - rtrentry, 34
- O_AUTHEN
 - iwconfig.h, 44
- O_ENCRYPT
 - iwconfig.h, 44
- O_NO_LP
 - iwconfig.h, 45
- O_NOT_DEFAULT_LP
 - iwconfig.h, 45
- O_SET_KEY
 - iwconfig.h, 45
- O_SET_MAC_ADDRESS
 - iwconfig.h, 45
- O_SET_TX_POWER
 - iwconfig.h, 45
- packets_recv
 - net_device_info, 30
- packets_sent
 - net_device_info, 30
- panid
 - net_device, 29
- payload
 - udp_recvfrom, 38
- PING
 - event.h, 42
- ping
 - icmp.h, 49
- ping_cbargs, 32
 - delay, 32
 - done, 33
 - error, 32
 - len, 32
 - ping_reply, 32
 - rsi, 32
 - src, 32
- ping_cbargs_t
 - icmp.h, 49
- ping_reply

- ping_cbargs, 32
- platform/avr/platform.h
 - __PLATFORM_H, 60
 - ASYNC_COUNT, 60
 - CPUSLEEP, 60
 - InterruptTypesEnum, 60
 - WATCHDOGTIMER, 60
- PROBE_UPDATE
 - svcs.h, 78
- READROUTE
 - route.h, 52
- REBOOT_CNTL
 - svcs.h, 79
- REBOOT_NOTIFY
 - svcs.h, 79
- RebootFlagsEnum
 - svcs.h, 77
- RECV
 - socket.h, 71
- recvbuf
 - tcp_event, 36
- recvlen
 - tcp_event, 36
- REPROGRAM
 - svcs.h, 78
- route.h
 - _ROUTE_H, 51
 - ADDRROUTE, 52
 - DELETEROUTE, 52
 - num_rtentries, 52
 - READROUTE, 52
 - RouteCmdsEnum, 51
 - routectl, 52
 - rtenry_t, 51
- RouteCmdsEnum
 - route.h, 51
- routectl
 - route.h, 52
- RouteFlagsEnum
 - svcs.h, 78
- ROUTER
 - svcs.h, 78
- RouterSvcCmdsEnum
 - svcs.h, 78
- ROUTESVC_CNTL
 - svcs.h, 79
- ROUTESVC_NOTIFY
 - svcs.h, 79
- rss_i
 - ping_cbargs, 32
- rtenry, 34
 - cost, 34
 - dst, 34
 - hops, 34
 - nxtHop, 34
- rtenry_t
 - route.h, 51
- s6_addr
 - in.h, 54
 - socket.h, 70
- s6_addr16
 - in.h, 54
 - socket.h, 70
- s6_addr32
 - in.h, 54
 - socket.h, 70
- SCH_CONTINUATION
 - event.h, 42
- sch_continuation
 - async.h, 64
- send
 - socket.h, 72
- sendto
 - socket.h, 73
- SETTIMER
 - event.h, 42
- setitimer
 - time.h, 83
- settimeofday
 - time.h, 83
- sin6_addr
 - sockaddr_in6, 35
- sin6_flags
 - sockaddr_in6, 35
- sin6_flowinfo
 - sockaddr_in6, 35
- sin6_port
 - sockaddr_in6, 35
- SOCK_DGRAM
 - socket.h, 70

- SOCK_DGRAM_FRAG
 - socket.h, 71
- SOCK_STREAM
 - socket.h, 70
- sockaddr_in6, 35
 - sin6_addr, 35
 - sin6_flags, 35
 - sin6_flowinfo, 35
 - sin6_port, 35
- sockaddr_in6_t
 - socket.h, 70
- socket
 - socket.h, 73
 - tcp_event, 36
- socket.h
 - _SOCKET_H, 70
 - ACCEPT, 71
 - accept, 71
 - AF_INET, 70
 - AF_INET6, 70
 - AFEnums, 70
 - CLOSED, 71
 - connect, 72
 - CONNECTED, 71
 - RECV, 71
 - s6_addr, 70
 - s6_addr16, 70
 - s6_addr32, 70
 - send, 72
 - sendto, 73
 - SOCK_DGRAM, 70
 - SOCK_DGRAM_FRAG, 71
 - SOCK_STREAM, 70
 - sockaddr_in6_t, 70
 - socket, 73
 - SocketTypeEnums, 70
 - tcp_event_t, 70
 - tcpbind, 74
 - TcpStatesEnum, 71
 - udp_rcvfrom_t, 70
 - udpbind, 75
- sockid
 - udp_rcvfrom, 38
- SocketTypeEnums
 - socket.h, 70
- src
 - ping_cbargs, 32
- START
 - svcs.h, 78
- STATUS
 - iwconfig.h, 45
- STOP
 - svcs.h, 78
- subtype
 - event, 23
- SUCCESS
 - errno.h, 40
- SvcCmdsEnum
 - svcs.h, 78
- SVCCNTL
 - event.h, 42
- svccntl
 - svcs.h, 79
- svcs.h
 - _SVCS_H, 77
 - CNTL_COUNT, 79
 - CPUPOWER_CNTL, 79
 - DOWNLOAD_UPDATE, 78
 - F_LEFT_NETWORK, 78
 - F_NETWORK_JOINED, 78
 - F_REBOOT_BY_GW, 78
 - F_REPROGRAM_BY_GW, 78
 - HOST, 78
 - NOTIFY_COUNT, 79
 - PROBE_UPDATE, 78
 - REBOOT_CNTL, 79
 - REBOOT_NOTIFY, 79
 - RebootFlagsEnum, 77
 - REPROGRAM, 78
 - RouteFlagsEnum, 78
 - ROUTER, 78
 - RouterSvcCmdsEnum, 78
 - ROUTESVC_CNTL, 79
 - ROUTESVC_NOTIFY, 79
 - START, 78
 - STOP, 78
 - SvcCmdsEnum, 78
 - svccntl, 79
 - SWUPDATE_CNTL, 79
 - SWUPDATE_NOTIFY, 79
 - SwUpdateCmdsEnum, 78
 - SystemSvcsEnum, 78

- SWUPDATE_CNTL
 - svcs.h, 79
- SWUPDATE_NOTIFY
 - svcs.h, 79
- SwUpdateCmdsEnum
 - svcs.h, 78
- SystemSvcsEnum
 - svcs.h, 78
- tcp_event, 36
 - addr, 36
 - rcvbuf, 36
 - rcvlen, 36
 - socket, 36
 - type, 36
- tcp_event_t
 - socket.h, 70
- TCPBIND
 - event.h, 42
- tcpbind
 - socket.h, 74
- TcpStatesEnum
 - socket.h, 71
- time.h
 - _TIME_H, 82
 - gettimeofday, 82
 - getttimer, 82
 - itimerval_t, 82
 - setitimer, 83
 - settimeofday, 83
 - timer, 84
 - timeval_t, 82
- timer
 - time.h, 84
- timeval, 37
 - tv_sec, 37
 - tv_usec, 37
- timeval_t
 - time.h, 82
- tv_sec
 - timeval, 37
- tv_usec
 - timeval, 37
- tx_pwr
 - net_device, 29
- type
 - event, 23
 - tcp_event, 36
- u6_addr16
 - in6_addr, 25
- u6_addr32
 - in6_addr, 25
- u6_addr8
 - in6_addr, 25
- udp_rcvfrom, 38
 - from, 38
 - len, 38
 - payload, 38
 - sockid, 38
- udp_rcvfrom_t
 - socket.h, 70
- UDPBIND
 - event.h, 42
- udpbind
 - socket.h, 75
- unistd.h
 - _UNISTD_H, 85
 - close, 85
- UP
 - iwconfig.h, 45
- v6_gaddr
 - net_device_info, 30
- v6_laddr
 - net_device_info, 30
- v6_prefix
 - net_device_info, 30
- WATCHDOGTIMER
 - platform/avr/platform.h, 60