CommunicationLibrary Documentation

Release v0.1.5

Embedded Systems Department University Duisburg-Essen

Nov 24, 2020

CONTENTS

CHAPTER

ONE

COMMUNICATIONMODULE

The CommunicationModule is a library aimed at 8bit-avr microcontrollers. The intent is to offer software support for several 802.15.4 based network chips like MRF24J40MA or the XBee. To allow interaction with these chips a drivers for the respective peripheral interface (USART, SPI) are included.

1.1 Dependencies

The library comes with very few but essential dependencies. To use the precompiled library you'll need

1. avr-gcc
2. CException
3. Mutex
4. PeripheralInterface

To use our build scripts you will have to install the Bazel build tool. You can find install instructions at https://docs.bazel.build/versions/0.26.0/install.html.

We recommend using the BazelCProjectCreator script to create your project. Additionally to the dependencies already present in the `WORKSPACE` file you will need to add the following lines:

```
http_archive(
    name = "EmbeddedUtilities",
    strip_prefix = "EmbeddedUtil-0.3.2",
    urls = ["https://github.com/es-ude/EmbeddedUtilities/archive/v0.3.2.tar.gz"],
)

http_archive(
    name = "PeripheralInterface",
    strip_prefix = "PeripheralInterface-0.7.1",
    urls = ["https://github.com/es-ude/PeripheralInterface/archive/v0.7.1.tar.gz"],
)

http_archive(
    name = "CommunicationModule",
    strip_prefix = "CommunicationLibrary-0.1.8",
    urls = ["https://github.com/es-ude/CommunicationLibrary/archive/v0.1.8.tar.gz"],
)
```

Alternatively you can copy the dependencies from `WORKSPACE` file in this project and add the `CommunicationModule` dependency (as shown above).

1.2 How to use the library

The library follows a strict separation of interfaces and implementation. Several different implementations of each interface may be in use at the same time (however keep in mind not to use the same physical ressource more than once). All implementation is hidden behind abstract data types. To start using a module you have to create the corresponding structs. To give users as much control over their memory usage as possible, every implementation offers two functions

1. `size_t InterfaceNameImplementationName_getADTSize(void);`
2. `InterfaceName InterfaceName_create(InterfaceName ptr_to_memory, OptionalConfigParameters parameters);`

The create function usually also initializes the implementation, so that after calling it you can start using the implementation. For details about functions offered by the interfaces see their doxygen documentation or take a look at their header files.

IMPORTANT: To use the library you will have to implement the `executeAtomically()` function. This is necessary to make the Mutex lib work. The corresponding header file can be found in the EmbeddedUtilities repo under `EmbeddedUtilities/Atomic.h`.

1.3 Predefined setups for the department's platforms

A ready to use setup for the Motherboard and ElasticNode hardware platforms is available. To compile these use the following command:

```
bazel build integration_tests:HardwareSetup --platforms @AvrToolchain//
↪platforms:Motherboard
```

or:

```
bazel build integration_tests:HardwareSetup --platforms @AvrToolchain//
↪platforms:ElasticNode
```

respectively.

1.4 How to use the setups in your own bazel project

Integrate e.g. the `MotherboardSetup` lib like so:

```
default_embedded_binary(
    name = "MyApp",
    srcrs = ["src/MyApp.c"],
    deps =  ["@CommunicationModule//Setup:MotherboardSetup"],
)
```

1.5 Exceptions

Instead of passing and handling error codes in long if-else statements, we use the CException library. However currently it is only partially used.

1.6 Known Issues

- non blocking functions are in development
- enabling promiscuous mode seems to prevent back to back reception of packages

CHAPTER

TWO

API

Driver Interface for 802.15.4 compliant hardware.

The *Mac802154* ADT is used to send/receive/analyze 802.15.4 packages. For performance and interoperability reasons the way message sending works is a little bit unintuitive. To prepare a message for sending you use several setter functions to set things like the destination address or the payload seperately. Once you are happy with your message setup, you can send the frame using the Mac802154_sendBlocking() function. IMPORTANT: To be unambiguous about the byte order, all addresses (including the pan id) are specified in the exact same order they are transferred with. This however might be different than how the address is presented in other software. E.g. in XCTU the address is presented in reversed byte order.

Typedefs

typedef struct *Mac802154* **Mac802154**

typedef struct *Mac802154Config* **Mac802154Config**

Enums

enum [anonymous]

Values:

enumerator FRAME_TYPE_BEACON

enumerator FRAME_TYPE_DATA

enumerator FRAME_TYPE_ACKNOWLEDGEMENT

enumerator FRAME_TYPE_MAC_COMMAND

enumerator ADDRESSING_MODE_NEITHER_PAN_NOR_ADDRESS_PRESENT

enumerator ADDRESSING_MODE_SHORT_ADDRESS

enumerator ADDRESSING_MODE_EXTENDED_ADDRESS

enumerator FRAME_VERSION_2015

enumerator FRAME_VERSION_2003

enumerator FRAME_VERSION_2006

Functions

void **Mac802154_configure** (*Mac802154* *self, **const** *Mac802154Config* *config)
This sets up internal fields and initializes hardware if necessary. The *Mac802154Config* struct can be safely removed after the function returned.

void **Mac802154_sendBlocking** (*Mac802154* *self)

void **Mac802154_setShortDestinationAddress** (*Mac802154* *self, **const** uint8_t *address)
A copy of the address is kept internally so you are free to delete it after function return. Be aware that all addresses have to be provided in network byte order

void **Mac802154_useExtendedSourceAddress** (*Mac802154* *self)
Call this to configure the module to include the extended source address when constructing a frame. Please note that you can currently only use either the extended or the short source address, but not both.

void **Mac802154_useShortSourceAddress** (*Mac802154* *self)
Call this to configure the module to include the short source address when constructing a frame. Please note that you can currently only use either the extended or the short source address, but not both.

void **Mac802154_enablePromiscuousMode** (*Mac802154* *self)
In promiscuous mode all 802.15.4 frames with a correct crc will be received, no matter how their address fields are set. WARNING: Remember to disable promiscuous mode before sending any packages. Otherwise your device might stop receiving any packages. (This has been observed for the MRF24J40.)

void **Mac802154_disablePromiscuousMode** (*Mac802154* *self)

void **Mac802154_setExtendedDestinationAddress** (*Mac802154* *self, **const** uint8_t *address)
sets address in big endian representation suitable for network transmission

void **Mac802154_setPayload** (*Mac802154* *self, **const** uint8_t *payload, size_t payload_length)
the payload needs to be alive in memory while transmission is running

uint8_t **Mac802154_getReceivedPacketSize** (*Mac802154* *self)

Return size of all data available, this might also include additional information like rssi

bool **Mac802154_newPacketAvailable** (*Mac802154* *self)
check whether the hardware has received new data since the last call to this function

void **Mac802154_fetchPacketBlocking** (*Mac802154* *self, uint8_t *buffer, uint8_t size)
Place all received data available from the hardware into the buffer.

Parameters

- `size`: This should be the value you got prior from *Mac802154_getReceivedPacketSize()*, generally you're free to use a different one but the result will almost certainly lead to problems with the inspection functions, that expect a complete frame.

const uint8_t *__Mac802154_getPacketPayload__ (*Mac802154* *self, **const** uint8_t *packet)

Return A pointer to the start of the payload field

uint8_t **Mac802154_getPacketPayloadSize** (*Mac802154* *self, **const** uint8_t *packet)

bool **Mac802154_packetAddressIsShort** (*Mac802154* *self, **const** uint8_t *packet)

bool **Mac802154_packetAddressIsLong** (*Mac802154* *self, **const** uint8_t *packet)

uint8_t **Mac802154_getPacketSourceAddressSize** (*Mac802154* *self, **const** uint8_t *packet)

const uint8_t *__Mac802154_getPacketExtendedSourceAddress__ (**const** *Mac802154* *self, **const** uint8_t *packet)

const uint8_t ***Mac802154_getPacketShortSourceAddress**(**const** *Mac802154* **self*, **const** uint8_t **packet*)

struct Mac802154Config

Public Members

uint8_t **short_source_address**[2]

uint8_t **extended_source_address**[8]

uint8_t **pan_id**[2]

uint8_t **channel**

struct Mac802154

#include <Mac802154.h> This struct below is exposed to allow developers providing new implementations of the interface. As well as to ease control of memory. Do not access the function pointers directly.

Public Members

void (***setShortDestinationAddress**)(*Mac802154* *self, **const** uint8_t *address)

void (***setExtendedDestinationAddress**)(*Mac802154* *self, **const** uint8_t *address)

void (***setPayload**)(*Mac802154* *self, **const** uint8_t *buffer, size_t size)

void (***useExtendedSourceAddress**)(*Mac802154* *self)

void (***useShortSourceAddress**)(*Mac802154* *self)

void (***sendBlocking**)(*Mac802154* *self)

void (***sendNonBlocking**)(*Mac802154* *self)

void (***reconfigure**)(*Mac802154* *self, **const** *Mac802154Config* *config)

uint8_t (***getReceivedPacketSize**)(*Mac802154* *self)

bool (***newPacketAvailable**)(*Mac802154* *self)

void (***fetchPacketBlocking**)(*Mac802154* *self, uint8_t *buffer, uint8_t size)

const uint8_t *(***getPacketPayload**)(**const** uint8_t *packet)

uint8_t (***getPacketPayloadSize**)(**const** uint8_t *packet)

bool (***packetAddressIsShort**)(**const** uint8_t *packet)

bool (***packetAddressIsExtended**)(**const** uint8_t *packet)

uint8_t (***getPacketSourceAddressSize**)(**const** uint8_t *packet)

const uint8_t *(***getPacketExtendedSourceAddress**)(**const** uint8_t *packet)

const uint8_t *(***getPacketShortSourceAddress**)(**const** uint8_t *packet)

void (***enablePromiscuousMode**)(*Mac802154* *self)

void (***disablePromiscuousMode**)(*Mac802154* *self)

INDEX

Symbols

M