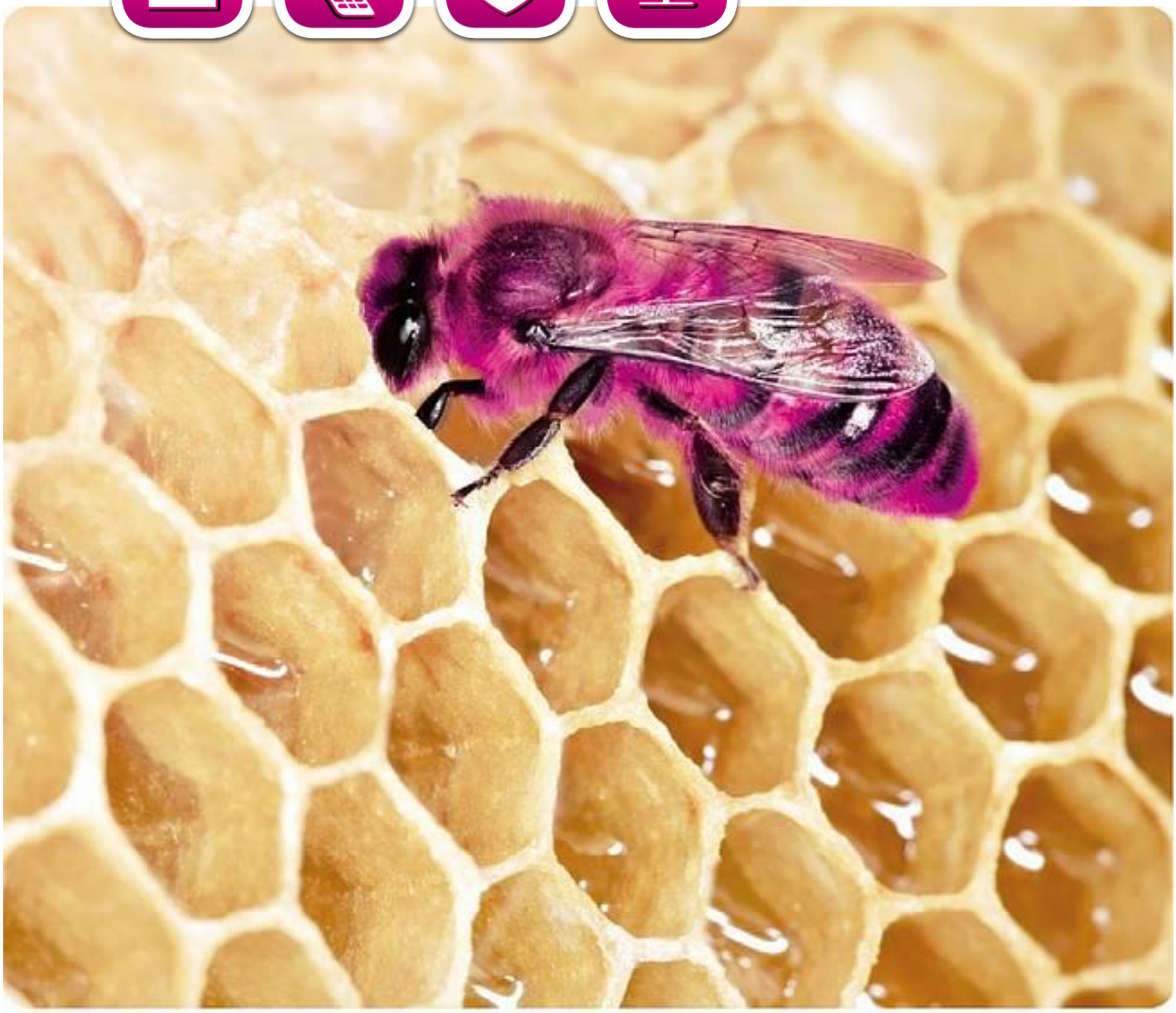
The Analog Mixed Signal Company

Programming Guide

Rev. 1.62 / August 2012

ZWIR451x

Low-Power Wireless IPv6 Module Series



Content

List of Figures

List of Tables

1 Introduction

This guide describes the usage of the 6LoWPAN application programming interface (API) for application development using ZWIR451x modules. These modules provide bidirectional IPv6 communication over an IEEE 802.15.4 wireless network. Using IPv6 as the network layer protocol allows easy integration of sensor or actor nodes into an existing Internet Protocol (IP) infrastructure without the need for additional hardware.

Figure 1.1 ZWIR451x-I Application Domain

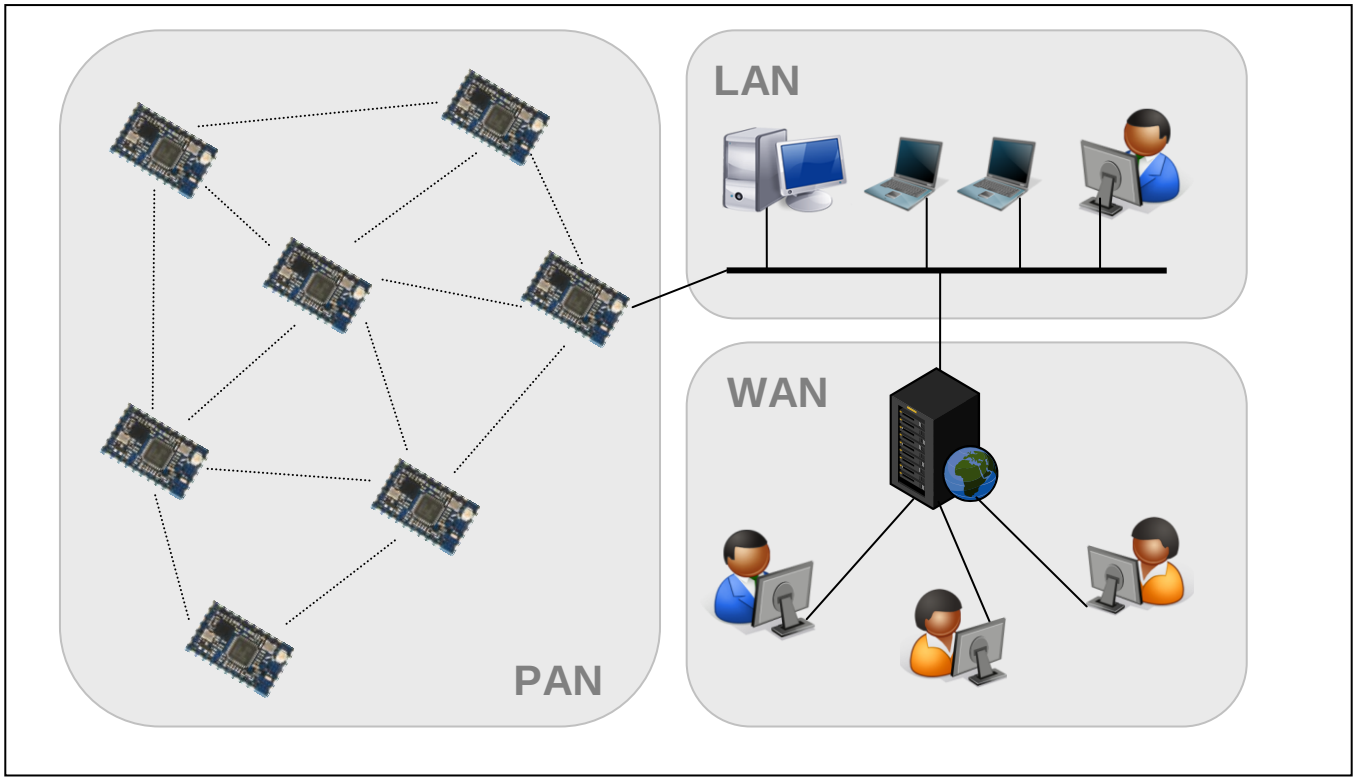


Figure 1.1 shows a typical network configuration. The Personal Area Network (PAN) is built from a set of various ZWIR451x modules. The network is connected via a border router to the local area network (LAN) and from there to a wide area network (WAN) such as the Internet. With this setup, each module can be accessed from anywhere in the world with just its unique IPv6 address.

The radio nodes are typically organized in a mesh topology. Routing of IP packets over this topology is handled by the software stack transparently for the user. The network allows dropping in new nodes or removing existing nodes without requiring manual reconfiguration. Routes to new nodes will be found automatically by the stack.

Application software runs on an ARM Cortex M3 microcontroller (MCU) on top of the ZWIR451x API. The MCU is clocked with up to 64 MHz and provides 256 kByte flash memory and 48 kByte RAM which allows the implementation of memory and computationally intensive applications. The API provides functions to communicate with remote devices, access different I/O interfaces and support power-saving modes.

1.1. IPv6

IPv6 is the successor of the IPv4 protocol, which has been the major network protocol used for Internet communication over the past decades. One of the main advantages of IPv6 over IPv4 is its huge address area, which provides 2^{128} (about 3.4 x 10^{38}) unique addresses. This enormous address space allows assignment of a globally unique IP address to every imaginable device that could be connected to the Internet. Another advantage with respect to sensor networks is the stateless address auto-configuration mechanism, allowing nodes to obtain a unique local or global IP address without requiring a specific server or manual configuration.

The use of IPv6 makes it possible to connect sensor networks directly to the Internet. Basically this is possible with other network protocols, too, but those require a dedicated gateway that translates network addresses to IP addresses and vice versa. Usually this translation requires application knowledge and maintenance of the application state in the border router, and therefore changing the border router software might be required with each application update. The protocol gateway might also introduce an additional point of attack if secure communication between devices inside and outside of the PAN is required.

ZMDI's 6LoWPAN implementation supports IPSec, which is the mandated standard for secure communication over IPv6. The use of IPv6 through the whole network allows real end-to-end security.

1.2. 6LoWPAN

IPv6 has been designed for high bandwidth internet infrastructure, which does not put significant constraints on the underlying network protocols due to the vast amount of memory, computing power and energy. In contrast, the IEEE 802.15.4 standard is intended for low data-rate communication of devices with very limited availability of all these resources. In order to make both standards work together, the 6LoWPAN standard (RFC 4944) has been developed by the Internet Engineering Task Force (IETF) to carry IP packets over IEEE 802.15.4 networks.

6LoWPAN adds an adaption layer between the link layer and network layer of the Open Systems Interconnection (OSI) reference model. This layer performs compression of IPv6 and higher layer headers as well as fragmentation to get large IPv6 packets transmitted over IEEE 802.15.4 networks. The 6LoWPAN layer is transparent for the user, and therefore on 6LoWPAN devices, the IPv6 protocol is used in exactly the same way as on native IPv6 devices. The presence of the 6LoWPAN adaption layer does not restrict IP functionality. The user of a 6LoWPAN system doesn't even recognize the existence of the 6LoWPAN layer.

1.3. Organization of this Document

The following section gives a system overview and shows the interfaces available for the programmer.

Sections 3 to 8 cover the API documentation, which is divided into two parts. The first part, covered by section 3 provides a functional description of the network stack. It explains the correlation of the different API functions and provides background information about stack internals. The second part is the function reference and is covered by sections 4 to 8. If familiar with the general stack functionality, the reader can just use these sections to look up function signatures or basic usage information.

Section 9 explains how user applications can use the resources provided by the microcontroller and which resources are blocked by the operating system.

Terms set in **`bold monospace`** font can be clicked, activating a hyperlink to the section where a detailed definition of this term can be found.

2 System Overview

ZMDI's ZWIR451x modules integrate an IEEE802.15.4 compliant transceiver (TRX) with a powerful ARM Cortex M3 microcontroller (MCU). The complete radio front-end is integrated. The transceiver performs analog and digital radio processing and implements parts of the medium access control (MAC) layer. The microcontroller implements the remaining part of the MAC and the higher protocol layers.

Figure 2.1 Schematic View of the ZWIR451x Module

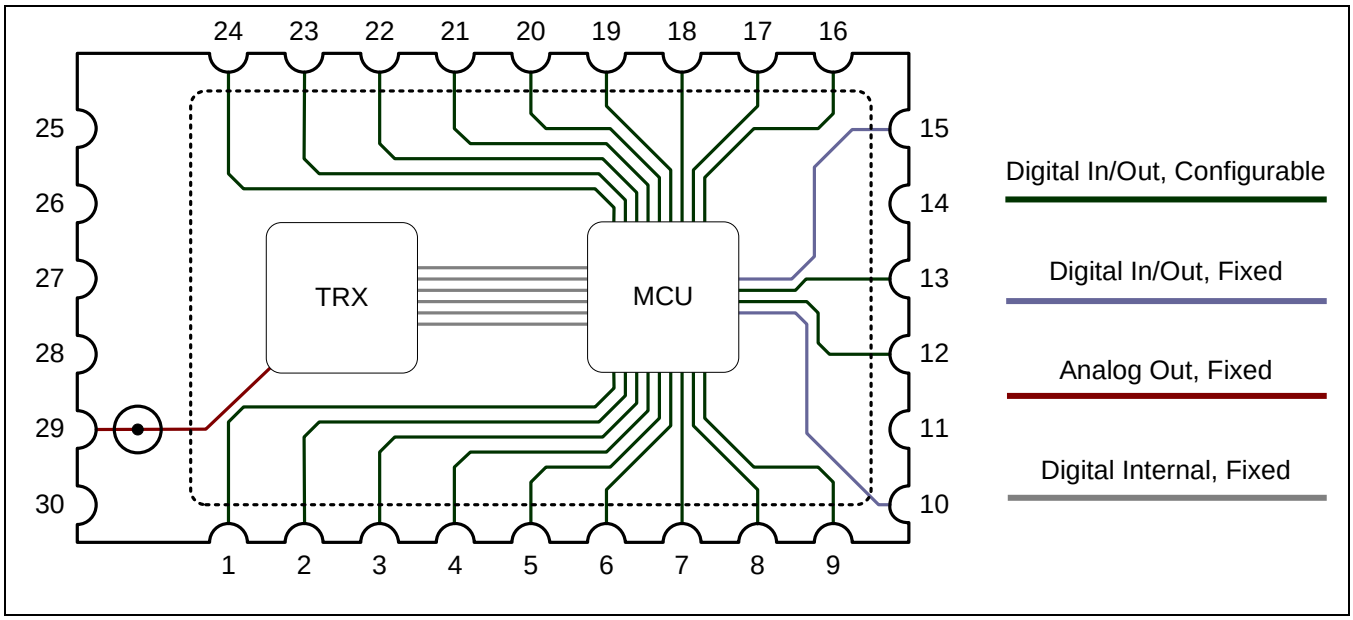


Figure 2.1 shows the outline of ZWIR451x modules with a schematic view of the internal components and connections. Users can disregard RF issues except the external antenna pin. Most of the external pins are general purpose digital pins that are connected directly to the MCU and are freely configurable. Table 2.1 lists the functionality of the pins and which port they are connected to on the MCU.

ZWIR451x modules are delivered with a 6LoWPAN stack, which is completely implemented on the MCU. The microcontroller provides enough resources to run a user application in parallel with the stack. The user application can make use of the rich set of peripherals, and system level costs are kept low because no external controller is required.

Table 2.1 Pin Description of ZWIR451x Modules

Pin	Name	MCU Port	Type 1	5V	Startup Function	Alternative Function
1	GPIO7	PA7	IO		GPIO (input mode)	SPI1 – MOSI ADC1 / ADC2 – IN7 PWM (TIM8-1N, TIM3-1, TIM1-1N)
2	GPIO6	PA6	IO		GPIO (input mode)	SPI1 – MISO ADC1 / ADC2 – IN6 PWM (TIM3-1) Timer Break (TIM1, TIM8)
3	GPIO5	PA5	IO		GPIO (input mode)	SPI1 – SCK DAC – OUT2 ADC1 / ADC2 – IN5
4	GPIO4	PA4	IO		GPIO (input mode)	SPI1 – NSS USART2 – CK DAC – OUT1 ADC1 / ADC2 – IN4
5	GPIO3	PA3	IO		GPIO (input mode)	USART2 – RX ADC1 / ADC2 / ADC3 – IN3 PWM (TIM2-4, TIM5-4)
6	GPIO2	PA2	IO		GPIO (input mode)	USART2 – TX ADC1 / ADC2 / ADC3 – IN2 PWM (TIM2-3, TIM5-3)
7	GPIO1	PA1	IO		GPIO (input mode)	USART2 – RTS ADC1 / ADC2 / ADC3 – IN1 PWM (TIM2-2, TIM5-2)
8	GPIO0	PA0-WKUP	IO		GPIO (input mode)	WKUP USART2 – CTS ADC1 / ADC2 / ADC3 – IN0 PWM (TIM2-1, TIM5-1) Timer Trigger (TIM2)
9	GPIO12	PC13	IO		GPIO (input mode)	TAMPER-RTC
10	/RESET	NRST	IO		Reset	Not available
11	GND	GND	S		Ground	Not available
12	GPIO9	PA10	IO	✓	GPIO (input mode)	USART1 – RX PWM (TIM1-3)
13	GPIO8	PA9	IO	✓	GPIO (input mode)	USART1 – TX PWM (TIM1-2)
14	VCC	VCC	S		Power Supply	Not available
15	BSEL	BOOT0	I		Boot mode selection	Not available

Pin	Name	MCU Port	Type [1]	5V	Startup Function	Alternative Function
16	GPIO10	PA11	IO	✓	GPIO (input mode)	USART1 – CTS USB – D- CAN – RX PWM (TIM1-4)
17	GPIO11	PA12	IO	✓	GPIO (input mode)	USART1 – RTS USB – D+ CAN – TX Timer Trigger (TIM1)
18	VSTDBY	VBAT	S		Alternative power supply for Standby Mode	Not available
19	TDO	PB3	IO	✓	JTAG – TDO	TRACESW SPI1 – SCK PWM (TIM2-2)
20	TMS	PA13	IO	✓	JTAG – TMS	GPIO 2
21	TDI	PA15	IO	✓	JTAG – TDI	GPIO 3 SPI1 – NSS Timer Trigger (TIM2)
22	TCK	PA14	IO	✓	JTAG – TCK	GPIO 2
23	GPIO14	PB7	IO	✓	GPIO (input mode)	I^2C – SDA UART1 – RX 4 PWM (TIM4-2)
24	GPIO13	PB6	IO	✓	GPIO (input mode)	I^2C-SCL UART1 – TX PWM (TIM4-1)
25	DIG1	-	O			Not available
26	PACTLN	-	O		PA control (differential) complementary output	Not available
27	PACTLP	-	O		PA control (differential)	Not available
28	GND	GND	S		Ground	Not available
29	ANT	-	IO		Antenna pin	Not available
30	GND	GND	S		Ground	Not available

1. *The "Type" column indicates the type of the pin: IO - input/output, I - input only, O - output only, S - power supply.*
2. *In order to enable alternative functions, field SWJ_CFG in MCU register AFIO_MAPR must be set to 0b100!*
3. *In order to enable alternative functions, field SWJ_CFG in MCU register AFIO_MAPR must be set to 0b010 or 0b100!*
4. *Remapped function.*

3 Functional Description

The following subsections give a generic overview of the different functionalities of the firmware delivered with ZWIR451x modules. Background information is provided if required for proper use of the libraries. Usage recommendations are given for optimal performance in certain application configurations. A detailed description of the functions, types and variables available for application programming is given in sections 4 through 8.

3.1. Requirements Notation

This document uses several words to indicate the requirements of ZMDI's 6LoWPAN stack implementation. The key-words *MUST*, *MUST NOT*, *REQUIRED*, *SHALL*, *SHALL NOT*, *SHOULD*, *SHOULD NOT*, *RECOMMENDED*, *MAY* and *OPTIONAL*, set in italic small caps letters denote requirements as described below

MUST, SHALL, REQUIRED These words denote an absolute requirement of the implementation. Disregarding these requirements will cause erroneous function of the system.

MUST NOT, SHALL NOT These phrases mean that something is absolutely prohibited by the implementation. Disregarding these requirements will cause erroneous function of the system.

SHOULD, RECOMMENDED These words describe best practice but there may be reasons to disregard it. Before ignoring this, implications of ignoring it must be fully understood.

SHOULD NOT, NOT RECOMMENDED These words describe items that, when implemented, can impair proper behavior of the system. However, there may be reasons to choose to implement the item anyway. Implications of doing so must be fully understood.

MAY, OPTIONAL These words describe items which are optional. No misbehavior is to be expected when these items are ignored.

3.2. Terms

This document distinguishes between three types of functions: hooks, callbacks and API functions. Basically, all three types are defined as normal functions in C, but they differ in the way that they are used.

API Functions are functions which are defined and implemented by ZMDI's 6LoWPAN stack. They provide a functionality that can be accessed by the user code. The declarations of API functions are provided in the header file belonging to the library the function is implemented in.

Hooks are functions that provide the user the ability to extend the default behavior of the stack. They are called from the operating system (OS) to give the application the opportunity to implement custom features or reactions to events. The operating system provides a default implementation of the hook that is called if no custom hook is defined. The prototypes of all available hooks are defined in the header file belonging to the library the default implementation is located in. **`ZWIR_AppInitNetwork`** and **`ZWIR_Error`** are examples of hooks.

Callbacks are also called from the operating system, but they need to be registered explicitly at the OS. The function may have a custom name, but the signature must be matching. Callback functions are registered at the OS using API functions. One example for a function expecting a callback is **`ZWIR_OpenSocket`**. In contrast to hooks, callbacks do not have default implementations. For each callback function there is a type declaration declaring how the signature of the user function should look like.

3.3. Naming Conventions

For better readability of the code, all user accessible functions and types of the API comply with a set of naming conventions. Each identifier that is an element of the API is prefixed with "**`ZWIR_`**." Function, variable, function argument and type identifiers are defined using "CamelCase" style. This means that each single word of a multiple word identifier starts with a capital letter. The remaining letters of the word are lower case. Preprocessor macros are defined using all capital letters.

Different style rules apply to functions, variables and types definitions. Function-name and type-name identifiers start with a capital letter in the first word, while variable identifiers start with a lower case letter in the first word. Type names have an additional "**`_t`**" suffix. Variable names and function arguments are not differentiated in the naming conventions.

Table 3.1 Naming Conventions Used in C-Code

Identifier Type	Style
`variableName, functionArgument`	First word starts with lower case, all other words with capital letters.
`FunctionName`	All words start with capital letters ("CamelCase").
`TypeName_t`	All words start with capital letters, "_t" suffix.
`PREPROCESSOR_MACRO`	All letters are capitalized.

3.4. Library Architecture

ZWIR451x modules are freely programmable by means of an API that is implemented in a set of libraries. The libraries provide different functionality and can be linked into the user program. The use of the core library is mandatory, as it provides the operating system and all generic communication functionality. All other libraries are optional and can be linked depending on the requirements of the target application. Each library exposes a set of functions and types that are required to implement the desired functionality. The library architecture is depicted in Figure 3.1.

To make programming as easy as possible, the libraries make use of an event and command approach wherever possible. Using this approach, application code is not required to poll for data on the different interfaces. Instead, newly available data is passed to user defined callback functions automatically. Timer hooks and callbacks are available, and they are executed periodically or after expiration of a user-defined time interval automatically.

Linking the library without any additional code will result in a valid binary that can be programmed on a radio module. Obviously such binaries will not provide user specific functionality. However, the nodes are relaying packets in mesh networks and are responding to ping requests. In order to add functionality, several functions that have empty default implementations can be defined by the user.

3.5. Operating Modes

The API provides three operating modes: Device Mode, Gateway Mode and Sniffer mode. The modes differ in how many of the protocol layers are processed by the network stack. All other API functionality remains the same. Setting the operating mode of a node must be done before any initialization of the API and the hardware. For this purpose, the **`ZWIR_SetOperatingMode`** function is provided.

Figure 3.2 shows how application code interfaces into the network stack in different operating modes. A description of the three different modes is provided in the next subsections.

Figure 3.1 Library Architecture

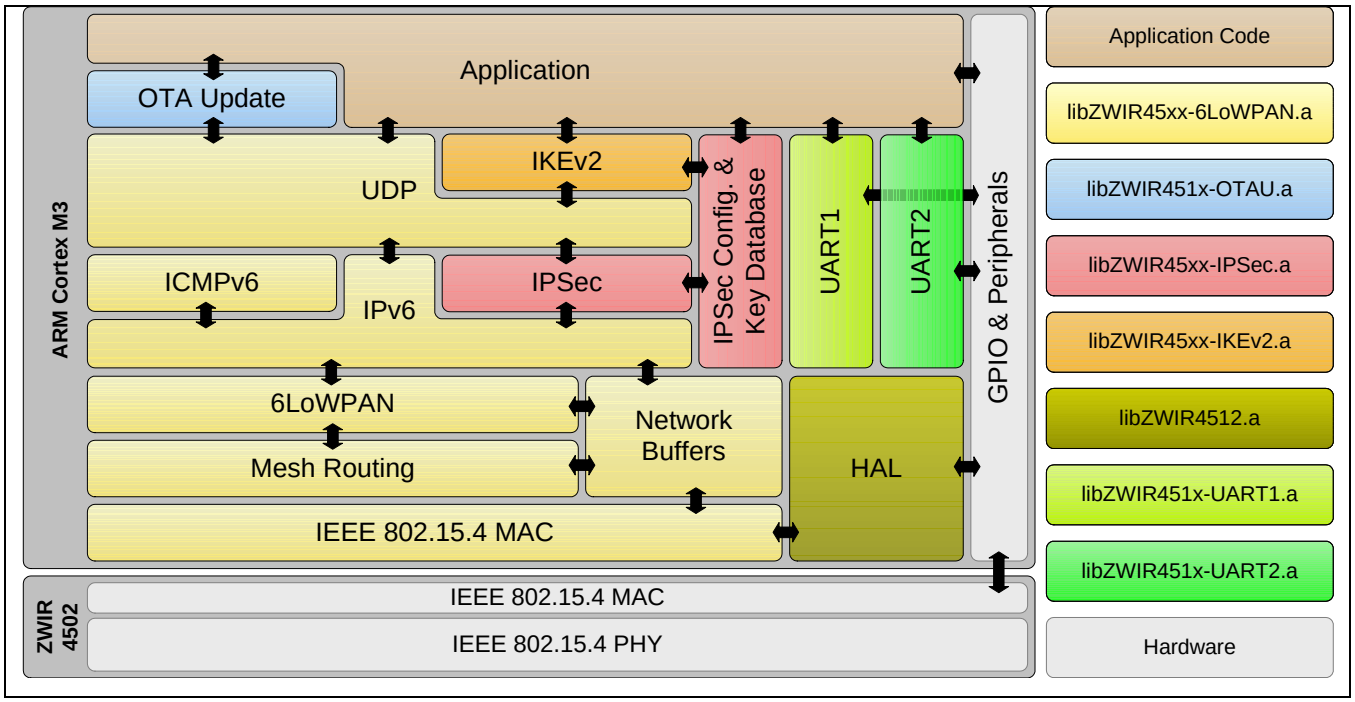
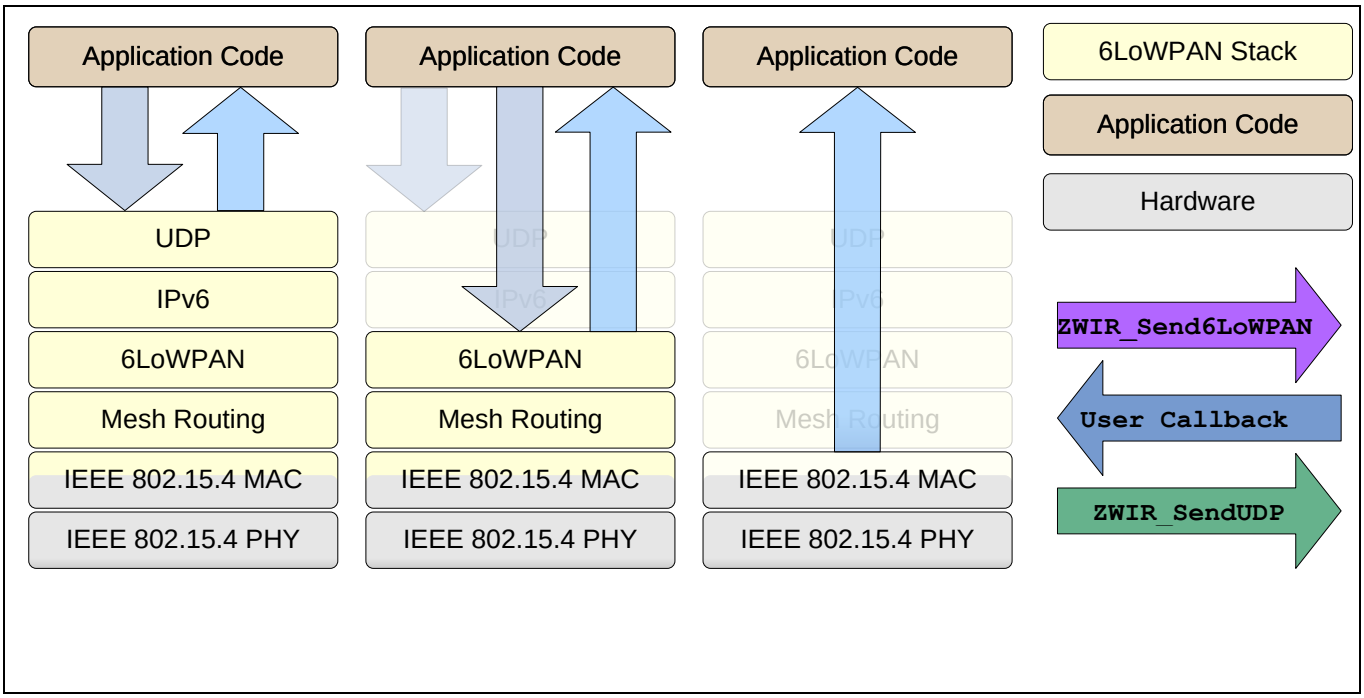


Figure 3.2 Application Interface into the Protocol Stack in Different Operating Modes



3.5.1. Device Mode

The Device Mode is preconfigured since this is the most commonly used mode for ZWIR451x modules. Each node with sensing or acting functionality should use this operating mode. Full protocol processing is performed for incoming and outgoing data. This means that all header information is removed from incoming User Datagram Protocol (UDP) packets and only payload data is passed to the application. Accordingly the application only has to provide payload data that should be sent over the network. The stack automatically adds all necessary header information.

Device Mode-configured devices behave as normal IPv6 devices. Therefore address auto-configuration and neighbor-discovery is performed as defined by the IPv6 standard. Data are sent and received over UDP sockets. The functions **`ZWIR_SendUDP`** and **`ZWIR_SendUDP2`** serve as an interface to the network stack. Incoming data is passed to an application callback that must be registered when a socket is opened using **`ZWIR_OpenSocket`**.

3.5.2. Gateway Mode

The Gateway Mode is intended for use with modules that should work as protocol gateways. Protocol gateways change the physical media used for IPv6 packet transmission. This enables the integration of 6LoWPAN networks into Ethernet-based IPv6 networks for instance.

In contrast to the Device Mode, not all network layers are processed in Gateway Mode. For any IPv6 packet that is received via the air interface, only the 6LoWPAN-specific modifications of the headers are removed, resulting in a packet containing all IPv6 and higher layer headers. This packet is passed to the receive callback function. Accordingly, all data that need to be sent over the network are assumed to have valid IPv6 and higher layer headers. Only 6LoWPAN-specific modifications will be applied to outgoing packets.

Gateway-configured devices do not perform address auto-configuration and neighbor-discovery as defined by the IPv6 standard. Moreover no router solicitation and router advertisement messages are generated automatically.

To enter the Gateway Mode, **`ZWIR_SetOperatingMode`** must be called from **`ZWIR_AppInitHardware`**. It is not possible to call **`ZWIR_SetOperatingMode`** from any other location in the code. **`ZWIR_SetOperatingMode`** accepts a callback function that is called upon reception of data in the gateway. Sending data is accomplished using the function **`ZWIR_Send6LoWPAN`**.

3.5.3. Sniffer Mode

The sniffer mode is provided to allow observation of raw network traffic. No protocol processing is performed. Thus the data passed to the application layer includes all header information. In contrast to the two other operating modes, all packets received over the air interface are passed to the application, regardless whether the packet which address they have been sent. This also includes MAC Layer packets.

Sniffer mode is useful for debugging purposes. It can be used to find out which devices in the network are transferring packets and which are not. Sniffer Mode devices do not generate network traffic at all, neither autonomously nor user triggered. That's why there is only an interface from the stack to the application code, but not vice versa.

To enter Sniffer Mode, call **`ZWIR_SetOperatingMode`** (***`ZWIR_omSniffer`***, ***`YourCallbackFunction`***) from **`ZWIR_AppInitHardware`**. The functions **`ZWIR_Send6LoWPAN`**, **`ZWIR_SendUDP`** and **`ZWIR_SendUDP2`** are not working in this mode. However, it is still possible to change the physical channel and the modulation scheme of the transceiver by calling **`ZWIR_SetChannel`** and **`ZWIR_SetModulation`**.

3.6. Operating System

The operating system is very light-weight and does not provide multi-threading. That means, any user defined function that is called from the operating system is completely executed before control is passed back to the operating system. Therefore, the user is required to write cooperative code. Users must be aware that functions requiring long execution time will block the operating system kernel and may cause the kernel to miss incoming data, regardless if they are received over the air or any wired interface.

3.6.1. Initialization

During the operating system initialization phase, the different libraries and the MCU peripherals required for system operation are initialized. Startup initialization is done in two phases, each of which has its own hook for user application code. During the first phase, the internal clocks and the peripherals used by the stack are initialized and the random number generator is seeded. Also peripherals required by certain libraries are initialized if the corresponding library is linked into the project. After this, the **`ZWIR_AppInitHardware`** hook is called if present, enabling application code to initialize further hardware. The application may initialize its I/Os and peripherals in this function. **`ZWIR_SetOperatingMode`** shall be called from here if Gateway Mode is required. Sending data over the network or initializing network sockets is not possible from here, as the network stack is not initialized. However, functions controlling the physical parameters of the network (e.g., output power or physical channel) should be called from here. Otherwise the first network operations that are done during initialization will be done with a possibly wrong parameter set.

During the second phase, the transceiver and the network stack are initialized. If the Normal Mode is selected, also duplicate address detection (DAD) is started and router information is solicited. DAD checks if the address given to the module is unique on the link. After finishing network initialization, **`ZWIR_AppInitNetwork`** is called. Application code may do its remaining initialization tasks such as setting up sockets here. Since DAD and router solicitation are started before the call to **`ZWIR_AppInitNetwork`**, it is recommended that physical parameters of the network are set up first in **`ZWIR_AppInitHardware`**. This will ensure that DAD and RS are performed on the correct channel with correct settings.

3.6.2. Normal Operation

During normal operation, the operating system collects events from the different peripherals and the application and handles them according to their priority. Event processing priorities are fixed and cannot be changed. Events are processed highest priority first; the lowest number represents the highest priority. Table 3.2 lists all events with their priorities and triggered actions.

Table 3.2 Event Processing Priority in the Main Event Loop

Priority	Event	Triggered By	Effect
0	Application Event 0	Application Code	Call user-defined callback function
1	Transceiver Event	Transceiver Interrupt Request	Process transceiver request
2	Application Event 1	Application Code	Call user-defined callback function
3	Callback Timer Expired	SysTick Controlled Software Timer	Call user-defined callback function
4	Sleep Requested	Software	Sleep for the requested time
5	Received Data on UART1	UART1 Interrupt	Call user-defined callback function
6	Application Event 2	Application Code	Call user-defined callback function
7	10 ms Timer Expired	SysTick Controlled Software Timer	Call **`ZWIR_Main10ms`**
8	100 ms Timer Expired	SysTick Controlled Software Timer	Call **`ZWIR_Main100ms`**
9	1000 ms Timer Expired	SysTick Controlled Software Timer	Call **`ZWIR_Main1000ms`**
10	Application Event 3	Application Code	Call user-defined callback function
11	Received Data on UART2	UART2 Interrupt	Call user-defined callback function
12	Sending Data Failed due to Resource Conflict	Network Stack	Retry sending
13	Application Event 4	Application Code	Call user-defined callback function

The operating system provides five application event handlers that can be used to process application events in the context of the operating system scheduler. Application event handlers should be used to react to asynchronous events requiring computationally intensive processing. Interrupts are a typical example for such events. If an interrupt occurs, the interrupt service routine (ISR) can trigger an event and delay the processing to an appropriate time. This ensures that multiple asynchronous events are handled in the order of their priority, without blocking interrupts.

Application events are triggered by calling **`ZWIR_TriggerAppEvent`** with the corresponding event number (0 through 4). When the OS scheduler reaches the user triggered event, an application callback function is executed. Multiple calls to this function before the corresponding application callback is invoked will not cause multiple invocations of the application callback.

For each application event, an event handler callback function must be registered using **`ZWIR_RegisterAppEventHandler`**. If no event handler is registered for a certain event, triggering this event has no effect. In order to change an event handler, **`ZWIR_RegisterAppEventHandler`** must be called again with the new handler. Unregistering event handlers can be performed by calling the registration function with a NULL callback argument.

3.6.3. Power Modes

The stack supports different modes to reduce the power consumption of the device. In Active Mode all module features are available. The Sleep, Stop and Standby Modes reduce the power consumption by disabling different module functionality. Each of the power-saving modes affects the behavior of the MCU and the transceiver and supports different wake-up conditions.

Table 3.3 Power Modes Overview

Mode	Wakeup		Clock		Context[1]	I/O	Transceiver
	Source	Time	MCU Core	Peripherals			
Active			On	On[2]	Retained	As Configured	On[3]
Sleep	Any IRQ	1.8 µs	Off	On[2]	Retained	As Configured	Off[4]
Stop	RTC IRQ External IRQ	5.4 µs	Off	Off	Retained	As Configured	Off[4]
Standby	RTC IRQ Wakeup Pin	50 µs	Off	Off	Lost	Analog Input	Off

1. *Refers to the status of the RAM and peripheral register contents after wakeup – the backup registers of the MCU are always available.*
2. *Clock is enabled for all peripherals that have been enabled by application code and all peripherals that are used by the library.*
3. *Can be powered off by application code.*
4. *Remains on if peripheral/transceiver is selected as wakeup source.*

Active Mode is entered automatically after startup. In this mode, the MCU core and all peripherals used by the application are running and all functionality is available. The transceiver is typically on, but can be switched off explicitly by a call to **`ZWIR_TransceiverOff`**. This mode has the highest power consumption.

In Sleep Mode, the MCU core is disabled but the MCU peripherals are still working if required. The transceiver can be switched on or off. Memory contents and I/O settings remain in the state that was active at the activation of the Sleep Mode. Waking up from Sleep Mode is possible on any MCU interrupt. After the wakeup event, the stack continues execution at the position it had been stopped. The power consumption in Sleep Mode is slightly reduced compared to Active Mode. If more significant reduction of the power consumption is required, the Stop or Standby Modes should be considered.

Stop Mode provides significant reduction of power consumption while still providing short wakeup time and context saving. Depending on the application's requirements, the transceiver may remain enabled to wake up the module when a packet comes in (set the transceiver as wakeup source). By default, the transceiver is disabled in Stop-Mode. The MCU core and all peripherals of the MCU are disabled in Stop Mode. Wakeup is only possible by the built-in RTC or an external interrupt, triggered at any GPIO line. For that, the external interrupt must be configured appropriately.

Standby Mode is the lowest power mode. In this mode, the MCU is powered off and the transceiver is on standby. Only the MCU's internal RTC is running, serving as a wakeup source. Additionally, the external wakeup pin can be used to wake up the module. After wakeup, the memory contents of the MCU are lost and must be reinitialized the same as after normal power-on.

Any of the low power modes is entered by calling the function **`ZWIR_PowerDown`**. It can be chosen whether power-down is delayed until all pending events are processes or not. If delayed power down is chosen, the power-down procedure can be aborted by a call to **`ZWIR_AbortPowerDown`**. The wakeup sources for the different power modes are configured by **`ZWIR_SetWakeupSource`**.

3.6.4. Error Handling

The stack performs error handling in two different ways. The first one is simply to reset the chip if an unrecoverable MCU exception occurs that caused an interrupt. For errors that are not caused by MCU exceptions, the stack provides a default handling which may be overwritten by the application code.

The error handlers performing a system reset are triggered by one of the interrupts listed in Table 3.4. The reason for resetting the whole system is that in the case of normal operation none of the listed interrupts should appear. However, if different behavior is desired, it is possible to overwrite the default implementation by providing own interrupt service routines. See section 10.2 for details.

Table 3.4 Interrupts Causing System Reset

Resetting Interrupts
Non-maskable Interrupt
Hard Fault
Memory Management
Bus Fault
Usage Fault
Programmable Voltage Detector

In the case of a recoverable error, the **`ZWIR_Error`** hook is called by the operating system. The error number is passed as function argument. In order to provide custom error handling the application MUST provide an implementation of **`ZWIR_Error`**. The return value of the function determines whether the error has been handled by the application (return **`true`**) or if the default handler shall be executed (return **`false`**).

3.7. Firmware Version Information

The ZWIR451x API provides the possibility of including firmware version information in the stack. This information can be requested remotely afterwards and are required by the Firmware Over-the-Air Update library. The complete firmware version consists of the Vendor ID, the Firmware ID, the Major Firmware Version, the Minor Firmware Version and the Firmware Version Extension. These components are defined in the application code using global variables. The role of the different components is explained in the following subsections.

Besides the firmware version information mentioned above, the stack provides additional version information for the library the application was linked with. This version information consists of Major Stack Version, Minor Stack Version and Stack Version Extension field.

3.7.1. Vendor ID

The Vendor ID is a 32 bit number which identifies the company that developed the device firmware. A Vendor ID must be requested from ZMDI. Each company must get its own Vendor ID before placing products on the market. The Vendor ID is set using the global variable **`ZWIR_vendorID`**. If this variable is not set, the firmware will use the Vendor ID $E966_H$, which is reserved for experimental purposes and must not be used for production firmware.

3.7.2. Product ID

The Product ID is a 16 bit number identifying the product firmware. It is especially important for the Over-the-Air Update functionality, but may also serve for remote identification of the device type. Refer to the application note “Enabling Firmware Over-the-Air Updates” for more information about the role of the Product ID in ZMDI’s Over-the-Air Update library.

The Product ID is set by defining the global variable **`ZWIR_productID`**. If this variable is not defined, the value will be read as zero.

3.7.3. Major Firmware Version

The Major Firmware Version is a version information field which is freely usable for application purposes. It is set by defining the global variable **`ZWIR_firmwareMajorVersion`**. If this variable is not defined the value will be read as zero.

3.7.4. Minor Firmware Version

The Minor Firmware Version is a version information field which is freely usable for application purposes. It is set by defining the global variable **`ZWIR_firmwareMinorVersion`**. If this variable is not defined the value will be read as zero.

3.7.5. Firmware Version Extension

The Firmware Version Extension is a version information field which is freely usable for application purposes. It is set by defining the global variable **`ZWIR_firmwareVersionExtension`**. If this variable is not defined the value will be read as zero.

3.7.6. Library Version

ZMDIs firmware stack libraries have their own version information included. This information is compiled into the binary libraries and may be requested by the application code using the function **`ZWIR_GetRevision`**. Like the firmware version, the library version consists of major and minor version as well as extension information.

3.8. Addressing

Each module has three types of addresses: a PAN identifier, link layer address and network layer address. This section describes the different address types and explains how they are used in the stack.

3.8.1. Address Types

The **PAN Identifier** (PANId) is a 16-bit-wide number carrying an identifier of the network. Each device in the same network must have the same PANId. Nodes with different PANIds cannot communicate. A default PANId is preprogrammed in the network stack. The current PAN Id can be requested or changed using the functions **`ZWIR_GetPANId`** and **`ZWIR_SetPANId`**, respectively. The default value is $ACCA_{HEX}$.

The **link layer address** is also referred to as the **MAC address** or **PAN address**. This address is used by the lower communication layers and does not need to be handled directly by the user. The link layer address must be unique in the network. Each ZWIR451x module has a predefined, hardware programmed address that is globally unique. The PAN address is 64-bits-wide. It can be requested and changed by the functions **`ZWIR_GetPANAddress`** and **`ZWIR_SetPANAddress`**. Changing the PAN address is not recommended as this could cause problems as described in section 3.8.4.

The third address type is the **network layer address**, which is equivalent to the **IPv6 Address**. These addresses are 128-bit-wide. They are used by the application to determine the destination that packets should be sent to or the source packets should be received from. Each device needs at least one IPv6 address to be reachable. However, multiple addresses can be assigned to each node. IPv6 addresses assigned to a node must be unique on the network. However, users typically do not need to handle Pv6 address assignment. IPv6 provides a mechanism that performs automatic address configuration. This mechanism is explained in section 3.8.3.

3.8.2. IPv6 Addresses

IPv6 addresses are 128-bit and therefore 16 bytes wide. As it would be impractical to use the byte-wise notation known from IPv4, IPv6 introduces a new notation. IPv6 addresses are represented by eight 16-bit hexadecimal segments that are separated by colons. An example for such address is

`2001:0db8:0000:0000:1b00:0000:0ae8:52f1`

The leading zeros of segments can be omitted as they do not carry information. Furthermore the IPv6 notation allows omitting a sequence of zero-segments and representing it as double colon. With these rules, the above address can be written as

`2001:db8::1b00:0:ae8:5211` or **`2001:db8:0:0:1b00::ae8:52f1.`**

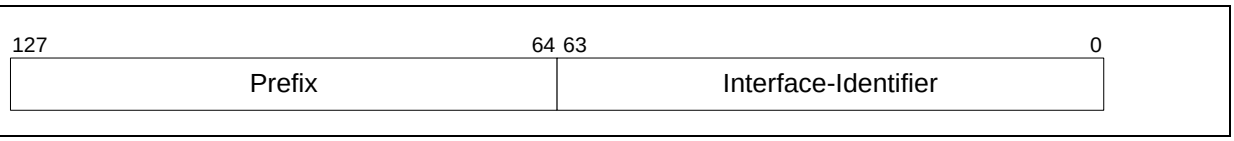
However, replacing multiple zero segments is not allowed, so the following address is invalid:

`2001:db8::1b00::ae8:5211`

An IPv6 address consists of two components: a prefix and an interface identifier. The prefix specifies the network a device is part of while the interface identifier specifies the interface of a device. A node with multiple network interfaces has multiple interface identifiers. The size of the prefix varies for different address types. In the IPv6 address notation, the prefix length can be appended to the address with a slash followed by the number of prefix bits. For example, the notation 2001:db8::/64 represents a network containing the addresses from 2001:db8:: to 2001:db8::ffff:ffff:ffff:ffff.

IPv6 supports three kinds of addressing methodologies: unicast addressing, multicast addressing and anycast addressing. Addresses for the different addressing schemes differ in how the prefix is formed.

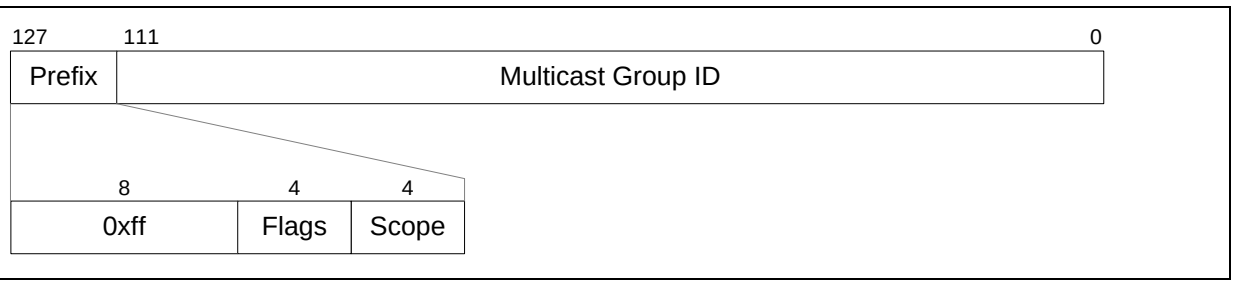
Figure 3.3 IPv6 Unicast Address Layout



Unicast addresses are shown in Figure 3.3 and use 64 bits each for the prefix and the interface identifier. Unicast addresses exactly identify one single interface in a network. The prefix of the address determines the scope of the unicast address. If the prefix equals fe80::/64 this is a link-local unicast address. Link local addresses are valid only on the single link a node is connected to. The prefix of global unicast addresses is received via address auto-configuration from a router that is connected to the Internet. There are additional prefix configurations with limited scope that are not covered by this documentation.

Multicast addressing allows sending out a single packet to multiple receivers. For this purpose, IPv6 provides multicast addresses. A multicast address can only be used as the destination address, but never as the source address of a packet. The layout of a multicast address is shown in Figure 3.4. Multicast addresses have a 16-bit prefix with the most significant 8 bits set to FF_{HEX}, followed by two 4-bit fields for flags and the scope of the multicast packet. The remaining bits specify the multicast group ID.

Figure 3.4 IPv6 Multicast Address Layout



In this document, it is assumed that the flags field is always either 0000_{BIN} or 0001_{BIN}. 0000_{BIN} specifies that the multicast address is a well-known address. 0001_{BIN} marks the address as a temporarily assigned address that is not specified by Internet standards. These addresses must be used for custom multicast addressing. The scope field is always assumed to be 0b0010, representing the link-local scope. Other scopes are usable but must be supported by routers.

Two specific addresses should be paid special attention, as these are used very often. More information about their use can be found in section 3.9.2.

1. The unspecified address ::
   All segments of this address are zero. It is used by receivers to listen to any sender. This address must never be used as destination address.
2. The link-local all nodes multicast address ff02::1
   Packets sent to this address are received by all nodes in the network, so this multicast address is equivalent to broadcasting.

For more detailed information about IPv6 addressing refer to RFC 4291 – “IP Version 6 Addressing Architecture”.

3.8.3. IPv6 Address Autoconfiguration

IPv6 provides a stateless address auto-configuration mechanism. This mechanism allows the configuration of node addresses from information being statically available on the node and information provided by routers. Router information is only required if global communication is required. Addresses for link-local communication can be derived from the link-layer address. This removes the need for manual configuration of addresses or dynamic host configuration protocol (DHCP) servers in the network.

The local information used for auto-configuration is the interface EUI-64 address. The EUI-64 address is a factory programmed link-layer address which ZMDI guarantees to be unique for each module. The EUI-64 address is often referred to as the MAC address. During network initialization, each node generates a unique link-local IPv6 address by putting the prefix fe80::/64 in front of the EUI-64 address with bit 1 of the most significant EUI-64 byte inverted. Assuming a link-layer address of 00:11:7d:00:12:34:56:78, the generated link-local IPv6 address would be fe80::211:7d00:1234:5678.

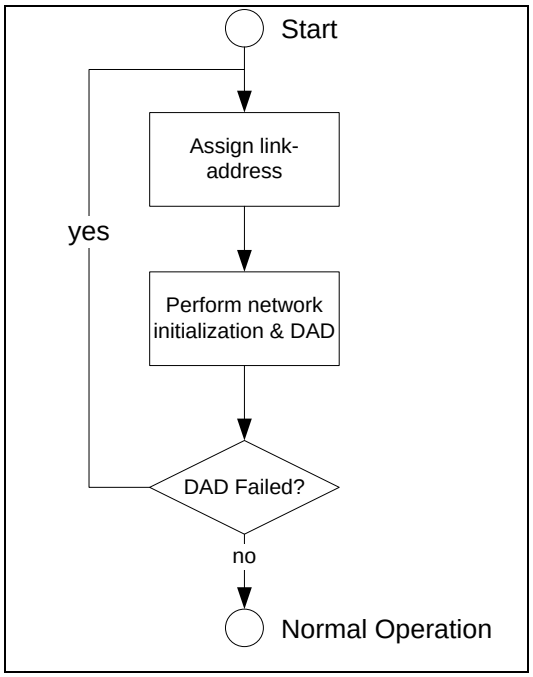
In addition to link-local address generation, nodes request router information during startup, trying to obtain a global prefix for building a globally valid address. Those requests are called router solicitations. Routers present on the link will respond to router solicitation messages of the node with router advertisements, containing global prefix information. Taking this prefix and the EUI-64 address of the node, a global address is generated in the same way as for the link-local address. Router solicitation is done automatically during the startup phase. If there is no router on the link, no global address will be assigned and only link-local communication is possible. In this case the router solicitation messages may be suppressed by setting the stack parameter **`ZWIR_spDoRouterSolicitation`** to zero.

A host cannot rely on a generated address to be unique, as there might be manually configured EUI-64 addresses on its link. Therefore, it must perform “duplicated address detection” (DAD) to be sure the generated address is unique. Duplicate address detection is mandatory for each address being attached to a node and is performed automatically by the network stack. It is described in more detail in the following section. Each address being assigned to an interface is subject to duplicate address detection. Addresses are not valid before duplicate address detection (DAD) is completed. Devices are not able to send or receive packets using an unicast address that has not been validated to be unique. After device startup (and therefore after assignment of the link-local unicast address), the network stack calls the hook **`ZWIR_AppInitNetworkDone`**, signaling that DAD on the link-local address has been completed and the address may be used. Applications should use this hook to send out initial packets.

3.8.4. Validation of Address Uniqueness

After a node has configured its own address it performs Duplicate Address Detection (DAD) to check if the newly configured IPv6 address is unique on the link. For this purpose, the node starts to send neighbor solicitation (NS) messages to the address to be checked (to its own address). If another node replies to one of those messages or if another node also sends neighbor solicitation messages to this address, the assigned address is not unique and must not be used. In this case, the error handler hook **`ZWIR_Error`** is called with the error code **`ZWIR_eDADFailed`**. It is up to the user to provide its own error handling mechanism for such cases. The default implementation provided in the library only removes the failing address from the interface. If the failing address was the only address of the module, the module will not be reachable.

Figure 3.5 Resolving Address Conflicts in Local Networks



Application code can try to resolve the address conflict. One possible solution is to manually change the link-layer address of the node, using random numbers or a dedicated algorithm. **`ZWIR_SetPANAddress`** must be called with the new address and the network initialization must be restarted. This is done by calling **`ZWIR_ResetNetwork`**. The procedure can be repeated for an arbitrary number of times until a unique address is found.

Note that a duplicate address problem should not appear if each module in the network uses the factory programmed link-layer address. In this case the link-layer address is guaranteed to be globally unique. Thus, it is recommended not to use the user's own addresses.

In some cases the application may be certain that there are no duplicate addresses in the network. In such cases the duplicate address detection mechanism may be disabled by setting the stack parameter **`ZWIR_spDoDuplicateAddressDetection`** to zero. This has the positive side effect of immediate ability to send and receive packets using the own IPv6 address(es). Furthermore, less traffic is generated on the network.

3.9. Data Transmission and Reception

Data are transmitted using the User Datagram Protocol (UDP). If a destination node is not directly reachable from the source node, packets are routed over intermediate nodes automatically. Route setup is done transparently for the user. The following subsections describe the different aspects of data transmission and reception.

3.9.1. User Datagram Protocol

The User Datagram Protocol is used for data communication. UDP is a connectionless and lightweight protocol, introducing minimal communication and processing overhead. No connection has to be created and no network traffic is required before data transmission between nodes can be started. Instead, communication is possible immediately. UDP does not guarantee that packets that have been sent are reaching the receiver. It is also possible that a single UDP packet is received multiple times. Furthermore, it is not guaranteed that the receiving order of packets at the destination is the same as the sending order at the source. This must be considered by the application programmer.

UDP uses the concept of ports to distinguish different data streams to a node. 65535 different ports can be distinguished in UDP. A port can be seen as the address of a service running on a node. Depending on the destination port of a packet, the network stack decides to which service the packet is routed on the receiver node. In ZMDI's 6LoWPAN stack, services, providing the callback functions that network packets are passed to, are running in the application code. Each service has it's own callback function.

3.9.2. Data Transmission and Reception

Data transmission is requested by calling **`ZWIR_SendUDP`** or **`ZWIR_SendUDP2`**. Both functions send a single UDP packet to a remote host. **`ZWIR_SendUDP2`** accepts the address and port of the remote device as a parameter, while **`ZWIR_SendUDP`** requires a socket handle, specifying the destination parameters. For reception of data, a socket is required as well.

A socket is an object that stores the address of a remote device and the remote and local UDP ports used for communication. It can be seen as an endpoint of a uni- or bi-directional communication flow. Additionally, a callback can be specified that is called when data is received over the socket. Sockets are opened and closed using the API functions **`ZWIR_OpenSocket`** and **`ZWIR_CloseSocket`**. The maximum number of sockets that can be open in parallel is defined by the stack parameter **`ZWIR_spMaxSocketCount`**.

Four parameters have to be provided when a socket is opened:

- IPv6 address of the remote communication endpoint: This is the address that data should be sent to and/or received from. Data reception is only possible if the remote address is a unicast address or the unspecified address. If a multicast address is provided, only data transmission is possible.
- Remote UDP port: This is the UDP port that data are sent to. For reception of data, this port is ignored.
- Local UDP port: This is the UDP port that data are received on. Only packets that are sent to this port will be handled in the callback function. If this number is 0, no data is received.
- Receive callback function: This is a pointer to a function that is called when data from a remote device is received. If no data should be received, this pointer can be set to NULL.

The choice of whether **`ZWIR_SendUDP`** or **`ZWIR_SendUDP2`** should be used for communication depends on the characteristics of the network traffic between the communicating devices. **`ZWIR_SendUDP2`** is intended to send few packets to a remote device without expecting a response from the target device. The function accepts the remote address and UPD port together with the data to be sent. Internally the function will open a temporary socket that is immediately closed after sending out the packet, so a slight overhead is added. **`ZWIR_SendUDP2`** functions even if the maximum number of sockets is open. **`ZWIR_SendUDP`** shall be used in cases where responses are expected from the remote device or data have to be transmitted frequently.

The following subsections will explain unicast and multicast communication in more detail and give examples of how to use the IPv6 addresses and ports appropriately.

3.9.2.1. Unicast

Traffic that has only a single destination node is called unicast traffic. In order to send data unicast, the sender must open a socket with the remote address set to the IPv6 address of the intended receiver. The receiver must open a socket with the remote address field set to the sender's IPv6 address or to the unspecified address. The sender socket remote port field must match the receiver socket local port field in both cases. Table 3.5 shows some example socket configurations and comments if communication is possible or not.

Table 3.5 Unicast Socket Examples

A)

fe80::1:1:1:1	
Rem. Addr. :	fe80::2:2:2:2
Rem. Port :	55555
Local Port :	44444

B)

fe80::2:2:2:2	
Rem. Addr. :	fe80::1:1:1:1
Rem. Port :	44444
Local Port :	55555

C)

fe80::2:2:2:2	
Rem. Addr. :	fe80::1:1:1:1
Rem. Port :	44444
Local Port :	33333

D)

fe80::2:2:2:2	
Rem. Addr. :	::
Rem. Port :	44444
Local Port :	55555

E)

fe80::3:3:3:3	
Rem. Addr. :	fe80::1:1:1:1
Rem. Port :	44444
Local Port :	55555

Sender	Comment
A	**B** receives packet. (Remote address and port of A match interface address and local port of B.) **C** does not receive packet. (Remote port of A does not match local port of C.) **D** receives packet. (Remote address of D matches all addresses; local port of D matches remote port of A.) **E** does not receive packet. (Remote address of A does not match interface address of E.)
B	**A** receives packet. (Remote address and port of B match interface address and local port of A.) No other socket receives packet. (Interface addresses do not match remote address field of B; local ports do not match remote port of B.)
C	**A** receives packet. (Remote address and port of B match interface address and local port of A.) No other socket receives packet. (Interface addresses do not match remote address field of B.)
D	No socket receives packet. (Sending is not possible with an unspecified address as the destination.)
E	No socket receives packet. (Remote address of sockets A, B, C do not match interface address of E; local port of D does not match remote port of E.)

3.9.2.2. Multicast

Multicast is used to send data to multiple nodes at the same time. For a multicast transmission, the sender must open a socket with the remote address set to a multicast IPv6 address. The semantics of ports is the same as for unicast communication. The receiver must open a socket with the remote address set to the IPv6 address of the sender or to the unspecified address. Note that a socket with a multicast remote address cannot be used for data reception.

ZMDI's implementation of the IPv6 multicast feature does not support explicit assignment of multicast groups to single nodes. Instead, if a packet is received that was sent to a temporary multicast address, the hook function **`ZWIR_CheckMulticastGroup`** is called by the network stack. This function must be implemented by application code that wants to make use of the multicast feature. The application can check the multicast group of the destination address and decide if it is part of it. This mechanism allows very flexible and application-tailored multicast addressing schemes. If the application does not provide the **`ZWIR_CheckMulticastGroup`**, temporary multicast addresses are rejected by the stack.

Table 3.6 Multicast Addressing Examples

A)

fe80::1:1:1:1	
Rem. Addr. :	ff02::1
Rem. Port :	55555
Local Port :	44444

B)

fe80::1:1:1:1	
Rem. Addr. :	ff02:x:x:x:x:x:x:x
Rem. Port :	55555
Local Port :	44444

C)

fe80::1:1:1:1	
Rem. Addr. :	ff12:x:x:x:x:x:x:x
Rem. Port :	55555
Local Port :	44444

D)

fe80::x:x:x:x	
Rem. Addr. :	fe80::1:1:1:1
Rem. Port :	x
Local Port :	55555

E)

fe80::x:x:x:x	
Rem. Addr. :	::
Rem. Port :	x
Local Port :	55555

Sender	Comment
A	D and E receive packet. (Remote address matches interface address of A; local port of D and E matches remote port of A.)
B	No socket receives packet. (If multicast group ID is not 1, packets are dropped by the receiver as well-known addresses must not be used by applications.)
C	D and E receive packet if multicast group ID resolution is implemented in user code and returns "true." (Remote address of C is temporary link-local multicast group; local ports of D and E match remote port of C.)

3.9.3. Address Resolution

Before unicast data can be transmitted from one device to another one, the sending node must determine the link-layer address of the receiver. This is called address resolution and is done automatically by the sender's network stack, using the neighbor discovery protocol (NDP). NDP replaces the address resolution protocol (ARP), which was used in IPv4 networks. Address resolution starts on demand and is transparent to the user.

If a data packet has to be sent to a receiver for which the link-layer address is not known, the sender performs address resolution to find the link-layer address of the receiver. The result is added to the so-called neighbor cache and the data packet is sent out. The maximum size of the neighbor cache is configurable using the stack parameter **`ZWIR_spNeighborCacheSize`**. Note that changing this parameter at runtime will result in the oss of all cache entries, regardless if the neighbor cache size is increased or decreased.

Neighbor cache entries are valid only for a limited time. After this time, the accessibility of the neighbor must be verified. This is also automatically by the NDP and is beyond the scope of this document. The lifetime of neighbor cache entries is defined by routers attached to the network. If the network doesn't have any routers, a default reachable time is used. In order to make this time configurable, ZMDIs network stack provides the stack parameter **`ZWIR_spNeighborReachableTime`**. In cases where routers advertise lifetime information, this information is given precedence over the stack parameter. Note that the default value for this constant significantly differs from the Ethernet default setting of 30 seconds. This is to reduce communication overhead generated by neighbor reachability detection messages. It is possible to disable the timeout completely. This is done by setting **`ZWIR_spNeighborReachableTime`** to zero.

The exact specification of the neighbor discovery protocol can be found in RFC 4861 – "Neighbor Discovery for IP version 6 (IPv6)".

3.9.4. Recommendations

The 6LoWPAN protocol performs IPv6 header compression to make transmission of IPv6 more efficiently transmittable over IEEE 802.15.4 based networks. The header compression mechanism is assuming that the interface identifier, thus the lower 64 bits of the IPv6 address, is generated from the link-layer address of the device. In such situations the header compression mechanism is capable of eliding link-local IPv6 addresses completely from the compressed header. Therefore, it is *NOT RECOMMENDED* using manually assigned IPv6 addresses. Instead the IPv6 addresses generated by address autoconfiguration after device startup *SHOULD* be used.

In order to achieve maximum compression of global IPv6 addresses it is possible to define compression contexts using the stack parameters **`ZWIR_spHeaderCompressionContext1-3.`** These parameters define frequently occurring prefixes which should be compressed by the 6LoWPAN header compression mechanism. If such prefixes are defined, it must be ensured that each device in the network uses the same configuration of these parameters!

In addition to IPv6 header compression, the 6LoWPAN layer may also compresses the UDP header. This is done if the source and/or destination port is in the range of 61616 to 61631. Thus, if the application doesn't explicitly require another port range, these ports *SHOULD* be used to maximize the data transmission efficiency.

3.10. Mesh Routing

ZMDI's 6LoWPAN stack enables devices to work in a mesh network topology. If the distance between two communicating devices is too big for direct radio transmission, packets are routed over intermediate devices – known as "hops" – automatically. Routes through the mesh are detected transparently for the application. Nodes may take two roles in a mesh network scenario: they may act as endpoints only, or they may provide relaying service. In this documentation devices are named endpoints or relays, depending on their configuration. If the stack configuration is not changed by the application each node is configured as relay with a maximum hop count of four.

ZMDI's mesh routing protocol is working on top of the MAC layer, just below the network layer.

3.10.1. Multicast Traffic

Network layer multicast traffic is handled by broadcast messages on the mesh and lower layers. Receivers of a mesh broadcast message may forward it, depending on their configuration. The decision whether the message is forwarded is taken based on the configuration parameter **`ZWIR_spMaxHopCount`**. This value actually determines the upper limit of hops a broadcast packet may take through the network. Each broadcast packet carries its hop count in its mesh routing headers. This field is incremented each time the packet is forwarded by a relay. When a node receives a packet with the hop count being equal to or bigger than **`ZWIR_spMaxHopCount`**, the node doesn't forward the packet. Otherwise the hop count is incremented and the packet is forwarded. Thus nodes can be forced to work as endpoints by setting **`ZWIR_spMaxHopCount`** to zero. Any other configuration makes a node a mesh network relay.

3.10.2. Unicast Traffic

For unicast traffic all nodes, hence endpoints and relays, maintain a so called routing table. The routing table stores the MAC address of the next hop to be taken to a certain destination MAC address. After power on, reset or network reset the routing table doesn't contain any entries. A node requiring unicast communication needs to set up a route to each of its unicast destination nodes. This is done on demand and transparent for the application.

When the transmission of a unicast packet is requested and there is no matching routing table entry for the destination, the packet to be sent is queued and the route discovery process is started. A Route Request (RReq) is broadcasted into the network, requesting a route to the destination address of the unicast packet. Nodes receiving a RReq check whether the requested address matches their own address or not. If not, the packet is retransmitted by relays and ignored by endpoints, respectively. If the own address is matched, a Route Reply (RRep) message is sent to source hop of the RReq packet. Nodes receiving a RRep packet create/update a record in their routing tables, storing the source address of the RRep packet as next hop to the requested destination.

Unicast packets always take the same route through the network as long as the route is not removed from the routing tables. Routing table entries are removed for one of the following reasons:

- The route has not been used for **`ZWIR_spRouteTimeout`** seconds
- The route has been failing for **`ZWIR_spRouteMaxFailCount`** times
- The route was oldest when a new route needed to be created but the routing table was full

If one hop is failing for **`ZWIR_spRouteMaxFailCount`** times (no acknowledge is sent by the hop), the sender considers the route as broken and sends an informative packet to the originator of the packet. The originator then reinitiates the route discovery process, searching for an alternative route.

3.10.3. Mesh Routing Parameter Configuration Recommendations

In order to maximize the network performance for different application scenarios while maintaining a high level of stability and without wasting resources, the different routing parameters should be configured according to the applications characteristics. Below all parameters are listed along with explanations of the basic function of the parameter and recommendations for their setting in different application scenarios.

ZWIR_spMaxHopCount

This parameter determines the whether a node acts as endpoint or as relay and constraints the forwarding of multicast packets. With **`ZWIR_spMaxHopCount`** set to zero the node acts as communication endpoint. Note that the node is still able to communicate with remote nodes over multiple hops. Only the ability to forward packets is constraint by this parameter!

In order to make a node a mesh network relay, **`ZWIR_spMaxHopCount`** *MUST* be set to a value greater than zero. However, the value *SHOULD NOT* be chosen arbitrarily but it *SHOULD* reflect the actual size of the network. The optimal value is the number hops required to reach the farthest remote communication partner. If no mesh routing is required, setting **`ZWIR_spMaxHopCount`** to zero will improve the performance.

It is strongly recommended limiting the number of nodes working as relay in the network. As a rule of thumb a relay should not have more than ten other relays in direct reachability. Otherwise the network latency and the packet loss are very likely to increase. If a large number of relays is desired, using the **`ZWIR_spRouteRequestMinRSSI`** parameter should be considered, to limit the amount of traffic generated during the route discovery process.

ZWIR_spRoutingTableSize

This parameter configures how many routes may be kept alive concurrently. Thus, this parameter defines with how many nodes the device may communicate without the need for dropping and reestablishing routes. The routing table is required in endpoints and relays! On endpoints the table size should be equal to or larger than the number of remote nodes the device wishes to communicate with. On relays this number should be increased by the number nodes relay service is provided for.

The routing table is stored in the RAM and therefore limited by the RAM size. The RAM for the routing table is quasi-statically allocated before the **`ZWIR_AppInitNetwork`** hook is called. Therefore it is recommended to define the size of the routing-table in **`ZWIR_AppInitHardware`**. Otherwise a network reset has to be performed in order to get the change into effect.

ZWIR_spRouteTimeout

This parameter defines how many seconds an idle route is kept in the routing table. The default value is 3600 seconds. The idle time counter is restarted each time the route is used. Typically the route timeout parameter doesn't explicitly affect memory consumption or application performance. However, in frequently changing network configurations reduction of the timeout value may be advantageous, as old routes don't have to be tested and found to be defect before a new route is established.

ZWIR_spRouteMaxFailCount

This parameter controls how often a route may fail before it is considered as dead. Depending on the network characteristics this value should be set to a rather low value between zero and five. The higher the probability of unreachability of a relay or endpoint, the lower this value should be selected. In networks with frequent changes of positions of nodes or a rapidly changing environment, the probability of unreachability is high and therefore, this variable should be low. In contrast, fixed installations of nodes and relays may select a higher value, as the unreachability of a node/relay is very likely to be temporary.

ZWIR_spRouteRequestAttempts

This parameter configures how many attempts are made to set up a route to a remote device. By reducing this number the application may reduce the network load caused by failing route discovery attempts. On the other hand reducing this number will increase the chance of a failing route discovery when it would be physically possible.

ZWIR_spRouteRequestMinLinkRSSI and ZWIR_spRouteRequestMinLinkRSSIReduction

Propagation of electromagnetic waves is influenced by a multitude of external parameters. As a result radio transmission sometimes appears to behave randomly. Typically this is caused by subtle changes in the external environment. The occurrence of random behavior may notably increase in mesh network topologies. For one logical connection of two nodes there are typically multiple physical links included, all of which have an independent failure probability.

In order to make links more robust against loss of connection due to environmental variations the parameters **`ZWIR_spRouteRequestMinLinkRSSI`** and **`ZWIR_spRouteRequestMinLinkRSSIReduction`** are provided. These parameters allow specifying link quality constraints on each physical link of the whole route. With such constraints in place, smaller environmental changes will impair the routes less, as the signal quality on any link is less likely dropping below the sensitivity level of the module.

3.11. Network and Device Status

The API provides functions for discovering the network and requesting the device status. Network discovery is performed using the **`ZWIR_DiscoverNetwork`** function. This function broadcasts a message to all devices in the PAN and makes the answers available to the user. For each device, the hop-distance, the link-quality and all assigned IPv6 addresses are returned.

The node status is returned by **`ZWIR_GetTRXStatistic`**. The returned data structure contains information such as sent and received packet and byte count and failing transmission attempts. However, the most important value is the sender duty cycle. This value is important, as frequency regulations require nodes to keep their transmission duty cycle lower than 1%. It is the responsibility of the application code to make sure that this number is not exceeded.

3.12. Security

Most applications require secure communication in order to protect sensitive data and to protect actors from unauthorized accesses through attackers. For that reason, ZMDI provides an implementation of the Internet Protocol Security Suite (IPSec) and the Internet Key Exchange protocol version 2 (IKEv2). IPSec is used to encrypt and authenticate data, while IKEv2 is used to manage the keys used for encryption and authentication. IPSec as well as IKEv2 are standardized by the Internet Engineering Task Force (IETF). Both protocols are mandated to be used for encryption and key management in IPv6. The implementation of the protocols is provided in two separate libraries.

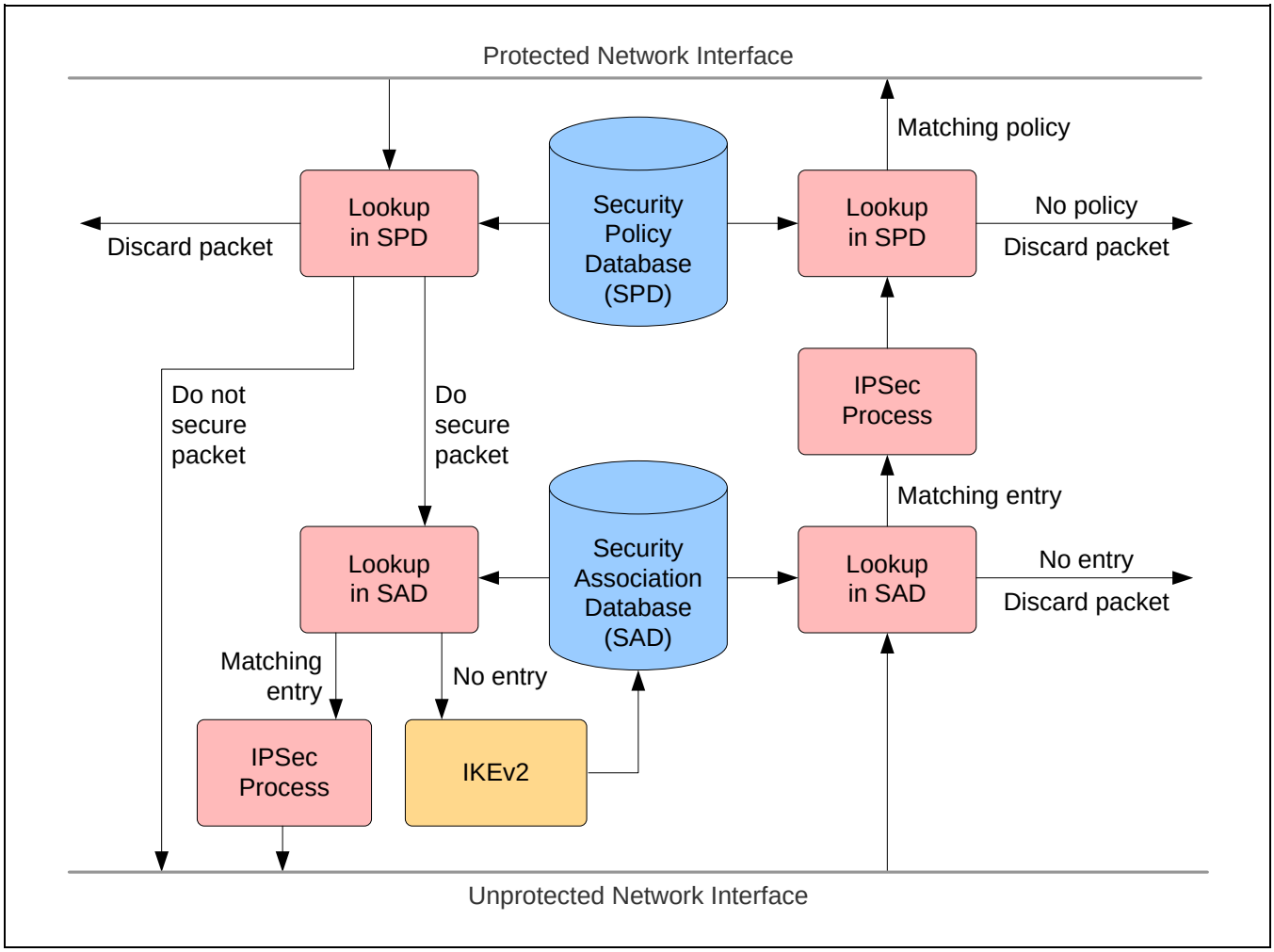
3.12.1. Internet Protocol Security (IPSec)

ZMDI provides an IPSec implementation in conjunction with its communication libraries. IPSec is a protocol suite for encryption and authentication of data sent over an IP network. IPSec is supported by virtually all modern operating systems. The encapsulating security payload (ESP) and authentication header (AH) protocols are supported for data encryption and authentication, respectively. Data encryption ensures confidentiality of information transmitted over the network. Authentication is applied to ensure that data are not modified along the way and that the sender is the entity that it claims to be.

In order to use the security features of the stack, the *libZWIR45xx-IPSec.a* library must be included in the project and must be configured appropriately. IPSec maintains a Security Policy Database (SPD) that contains rules for how outgoing and incoming traffic must be handled. For each incoming and outgoing packet, the stack checks the SPD for a matching rule that contains information on how the packet should be handled. The rules can direct the network stack to either drop, bypass or process the packet in the security module. Bypassing a packet means that no security processing is applied. Rules can be applied to single addresses or complete subnets.

Each item in the SPD requiring security processing contains a pointer into the Security Association Database (SAD). Each item of this database contains the required information for encryption and decryption of packets. This information includes keying material and the algorithm to be used for encryption and decryption.

Figure 3.6 Working Principle of IPSec



The SAD items can be configured manually or automatically. For automatic configuration, the Internet Key Exchange protocol is used. This protocol is implemented in a separate library and is described in the following section. In either case, SPD entries for incoming and outgoing traffic must be configured by the application. This is done using the function **`ZWIRSEC_AddSecurityPolicy`**. If manual configuration should be used, the function **`ZWIRSEC_AddSecurityAssociation`** must be called on both communicating devices, setting the security parameters for the connection.

3.12.2. Internet Key Exchange Version 2 (IKEv2)

The Internet Key Exchange version 2 protocol can be used for automatic creation of keying material for secured connections. This protocol is implemented in the *libZWIR45xx-IKEv2.a* library. If IKEv2 is used, no manual configuration of the SAD is required. Instead, keying material is negotiated automatically on demand. If application code tries to send data to a remote node and the according SPD entry requires security processing of this data, it is checked whether a security association is assigned to the SPD entry. If no such entry exists, IPSec requests the establishment of a security association from the IKEv2 daemon.

IKEv2 first tries to set up a secure communication channel over which keying material is exchanged. This channel is set up using the Diffie-Hellmann-Key-Exchange algorithm. Both communicating parties use a Pre-Shared Key (PSK) for mutual authentication. The PSK is registered using the **`ZWIRSEC_AddIKEAuthenticationEntry`** function. After setup of the secure channel, keying material for the security association to be created is exchanged.

3.12.3. Recommendations

ZMDI strongly recommends using the security features provided by the network stack. Security is not only required to prevent data from being visible for third parties. More critical are active attacks on a network. Most applications will suffer from such attacks. In the best case, applications may be behaving erratically; however, in the worst case, perilous behavior of actors can be caused by an attack. Attacking possibilities are manifold: Packets can be changed on their way to the destination; they can be sent again; invalid packets can be infiltrated into the network; or packets can be simply blocked by an attacker. IPSec can protect against all of these attacks. IPSec in conjunction with IKEv2 further increases the security, as the keying material can be renewed on a regular basis.

3.13. Firmware Over-the-Air Updates

ZMDI provides an over-the-air update (OTAU) library. This library extends the application with functionality for the reception and processing of update packets, as well as functionality for replacing the existing code with a new version. The update mechanism incorporates recovery mechanisms, ensuring the proper recovery after occurrence of an error during the update process.

The OTAU firmware library is designed to require minimal interaction of the firmware programmer. However, it places some constraints on the firmware in order to ensure reliability of the OTAU function.

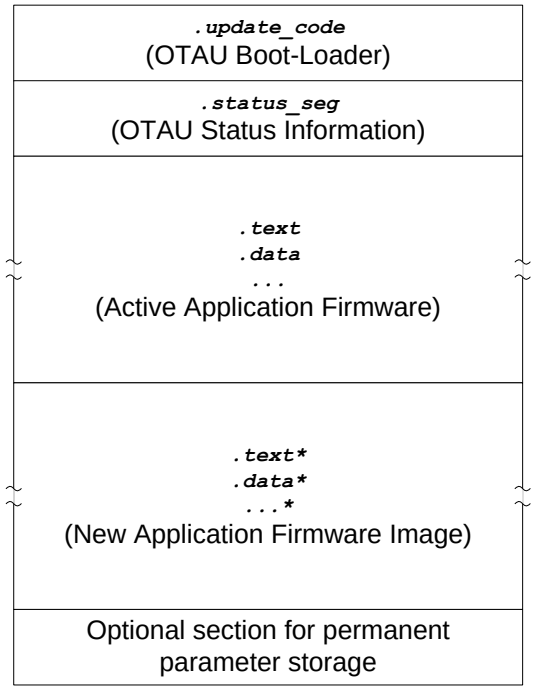
3.13.1. Functional Description

Integrating the firmware over-the-air update (OTAU) adds two components into the user application. The first one is a service for the reception and processing OTAU-related network traffic. The second one is a boot-loader that replaces the old firmware image with the new one after complete reception and verification of all update traffic. The boot-loader component is located in a program section called ***`.update_code`***. The location of this segment *MUST* be the first flash memory pages(s). During the update process, the boot-loader is not replaced! A second segment that is dedicated to OTAU-enabled code is the ***`.status_seg`***. This segment resides directly behind the boot-loader component and stores status information about the firmware update.

Including the sections referenced above, the application's memory layout would be as shown in Figure 3.7. The application code is located after the ***`.update_code`*** and ***`.status_seg`*** sections. Optionally the OTAU memory layout may incorporate a section for the storage of permanent parameters. This section MUST be located at the end of the flash. In contrast to non-OTAU-enabled applications, the amount of memory available for the application is limited to less than one half of the microcontroller's total flash memory size. This is because the space for the buffering of the full new firmware image must be provided.

The OTAU network service is started through a call to the **`ZWIR_OTAU_Register`** function. The function takes the UDP port to be used by the OTAU network service as the argument. Calling this function is all that is required to enable the reception and processing of firmware over-the-air updates. All other update parameters are controlled by the update server, which is typically a computer in the network.

Figure 3.7 Memory Layout of OTAU-Enabled Applications



A new firmware image to be loaded into the device is typically received in small chunks of data. Whenever a chunk of data is received, the corresponding flash location in the new application firmware image portion of the flash is updated. If this is the first chunk written to a flash page, the flash page is completely erased before the chunk is written to it. All packets corresponding to the same firmware update MUST include the same version information and the size of data chunks must be the same for all packets. Packets containing fragments of different size than the first fragment are ignored, once the update is started. Packets containing different version information trigger a complete re-initialization of the update.

3.13.2. Firmware Constraints

The OTAU network service uses a dedicated UDP port for the reception and transmission of OTAU-related packets. The application *MUST NOT* use the same port for any other purpose. If this limitation is ignored, the application behavior is determined by the behavior of sockets being reopened with the same parameters.

The contents of the ***.update_code*** and ***.status_seg*** segments must not be changed in any way by the application. This means the application *MUST NOT* explicitly place functions or data in either of these sections!

In order to allow the OTAU from a firmware-version ***A*** to a firmware version ***B***, the firmware versions MUST share the following properties:

- The call stack of A and B must be located at the same RAM position
- The call stack of A and B must be of the same size
- The flash memory layout of A and B must be the same (e.g., no optional section as shown in Figure 3.7 can be added or removed)

3.14. Memory Considerations

Applications must take care of their memory consumption – especially with respect to RAM (random access memory). There are basically three components which contribute to the overall RAM size requirement of the application:

1. Statically allocated memory
2. Dynamically allocated memory
3. Call-Stack

The call stack is required by the application to store the return addresses from function calls, function arguments and variables stored locally in functions. The RAM size reserved for the call-stack can be configured in the linker-script. Applications utilizing ZMDIs network stack need a minimum of 2 kB of call-stack. In order to leave some flexibility for user application, the default stack size configured in the linker script is 5 kB. The call stack resides at the lower end of the RAM area.

Static memory is consumed by globally declared and local statically declared variables. ZMDIs network stack requires less than 8 kB of static memory in a minimal configuration. The static memory consumption can easily be found out by examining the map-file generated by the linker. Static memory is allocated right behind the call stack.

The third component, the dynamically allocated memory, is allocated in the unused area between static memory and the end of RAM. Memory in this area is typically allocated at runtime using C's **`malloc`** function. Memory may be freed using the **`free`** function. Some lists and buffers used by ZMDIs network stack are allocated dynamically. There is no tool support for automatic determination of the dynamic memory size requirements of applications. The size of dynamic memory used by the ZMDIs network stack depends on the configured parameters. This is explained in more detail in section 3.14.2.

3.14.1. Call Stack

ZMDI's network stack places the Call-Stack of the application at the lower end of the RAM. This is to be able to detect stack overflows. The stack grows downwards from its topmost address towards the beginning of the RAM. When a stack overflow occurs, the MCU tries to access an address which is not in the RAM area and the MCU will generate a Bus-Fault interrupt. During interrupt handling, it must be considered that the interrupt handler function doesn't have a working stack. Thus, it is not secure to use local variables or calling subroutines. The Bus-Fault interrupt default handler performs a system reset.

3.14.2. ZMDI Network Stack Dynamic RAM Requirements

ZMDI's network stack has a number of configurable parameters which require allocation of memory at runtime. During system startup and after reset memory for these variables is allocated dynamically on the Heap. Table 3.7 gives hints on how these parameters influence the dynamic RAM requirements of the application.

Table 3.7 Stack-Parameter Dynamic Memory Size Requriements

Parameter	Size[1]	Element Count		Comment
	[bytes]	Min	Default	
ZWIR_spRoutingTableSize	28	1	8	
ZWIR_spNeighborCacheSize	60	1	8	
ZWIR_spMaxSocketCount	28	4	8	
-	238	-	6	This memory is allocated for internal buffers whose size cannot be configured
-	48	-	2	

1. *This column specifies the size of a single element. It must be multiplied with the configured parameter value. For each row an additional 32 byte element is required for storing the allocation record, if not otherwise denoted.*

Besides the quasi-statically allocated memories above, ZMDI's network stack dynamically allocates memory at runtime if packets have to be sent to destinations for which address resolution and route discovery have not been performed. One packet can be buffered for each destination node. Allocation is performed if enough memory is available. If no memory is left, the packet to be sent is dropped, but the address resolution and route detection procedure is initiated anyway. Thus, even if the packet is not being buffered, the next packet being sent is likely to arrive at the destination node.

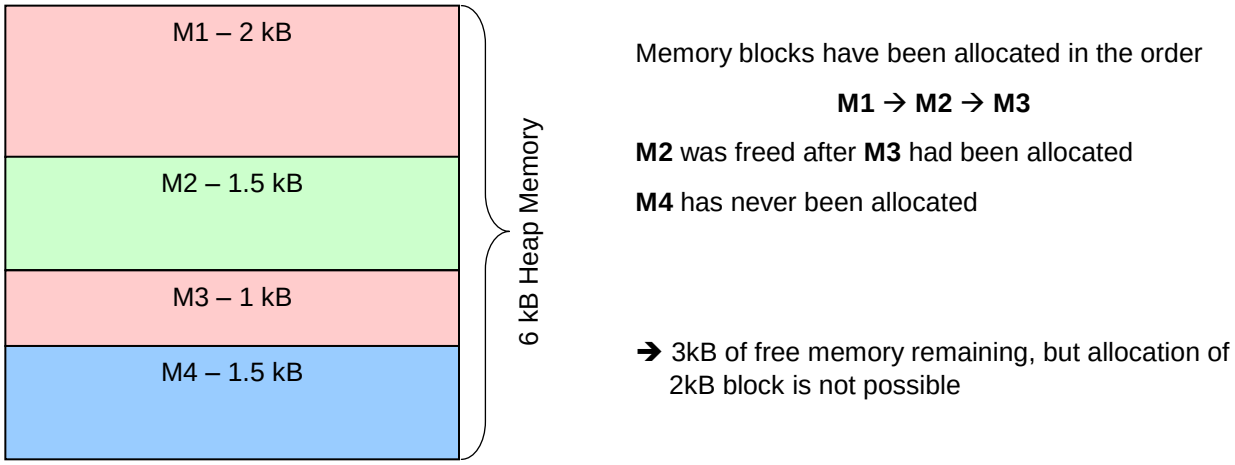
Application developers always have to make sure that the parameter settings allow proper allocation of all quasi-static memory. If parameters are chosen too big, stack initialization will fail and report the error **ZWIR_eMemoryExhaustion**. The default handling of this error is a system reset. Thus, if the parameters causing the memory exhaustion are set during system startup, this will result in an infinitive loop. The error is detected easily with a custom implementation of the **ZWIR_Error** function.

3.14.3. Using Dynamic Memory Allocation

Applications requiring dynamic memory allocation may freely use the function **malloc** and **free** for allocation and deallocation of RAM at runtime, respectively. However, the application developers must be aware of the limited availability of RAM on the device. Each allocated block consumes an additional 32 byte block on the heap as allocation record.

Due to memory fragmentation effects, it cannot be guaranteed that memory allocation is successful, even if the total amount of free heap space is sufficient for an allocation request. If memory blocks are allocated and freed frequently, it may happen that the free space in memory is scattered over the whole heap, not providing any free block big enough for holding a requested block. Figure 3.8 demonstrates this with a simple example which has 3kB of free memory but doesn't allow the allocation of a 2kB memory block with **malloc**.

Figure 3.8 Heap Memory Scattering



3.15. Supported Network Standards

Table 3.8 lists RFCs that are supported by ZMDI's network stack, and it specifies the limitations that apply with respect to these RFCs.

Table 3.8 Supported RFCs and Limitations

RFC	Limitations
Internet Protocol Version 6 (IPv6) Specification	

2460	• Hop-by-hop options header
	  ○ Only Pad1 and PadN options are supported (as specified in RFC).
	  ○ Other options will cause unrecognized option processing as proposed in RFC.
	• Routing extension header
	  ○ If segments left > 0, packets are ignored and icmp error message parameter problem is sent. However, the use of the type 0 routing header has been deprecated by RFC 5095!
	• Fragmentation extension header
	  ○ Not supported – packets received with this extension header are silently dropped.
	  ○ Spec requirement of receiving 1500 byte packets is not supported. However use of fragmentation is discouraged by the RFC!
	• Destination options header
	  ○ Only Pad1 and PadN options are supported (as specified in RFC).
	  ○ Other options will cause unrecognized option processing as proposed in RFC.
	  ○ Packets with next header 59 are dropped.
	  ○ Traffic class and flow label are always set to zero in packets sent from 6LoWPAN nodes.

Security Architecture for the Internet Protocol	
4301	• Tunnel mode is not supported. • ESP SA with both null encryption and no integrity algorithm is allowed. • Events are not logged. • Local IPv6 address cannot be used as selector. • No Sequence Counter Overflow. • SA lifetime is handled by IKE and only time controlled. • Certificates are not supported for IKE authentication. • No ICMP error messages processing and generation. • Fragmentation and reassembly is not supported.
IP Authentication Header	
4302	• AH is not supported.
IP Encapsulating Security Payload (ESP)	
4303	• SPI 0 to 255 are not reserved. • Anti-replay service is not active. • The sequence number will cycle. • ESN is not supported. • Dummy packets are not supported. • TFC Padding is not supported. • Auditing is not supported.
Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH)	
4835	• Supports ONLY NULL-encryption and AES-CTR. • Supports ONLY NULL-authentication and AES-XCBC-MAC-96.
Neighbor Discovery for IP Version 6 (IPv6)	
4861	• Only host functionality is implemented. • Destination cache as proposed by the standard is not available. However, no need for this as this should speed up next hop determination, which is done in a fraction of the time that message transport requires. • No checking of the linkMTU option is performed – however this is not required as 6LoWPAN only supports the minimum linkMTU of 1280 and will never send larger packets. • If no reachable router is in the router list, default router selection is not performed in a round-robin manner as proposed by the standard, but the first entry found is taken. However, reachable routers still have precedence over routers whose reachability is unknown (Section 6.3.6). • Variables are mainly implemented as constants. • During address resolution packets must be queued until address resolution is complete. The memory for this is allocated dynamically at runtime. If memory allocation fails, the packet is not being queued! Only one single packet will be queued. A queued packet is not replaced with the latest one if more than one packet needs to be buffered during the address resolution process. • Changes of the link-layer address are not advertised as proposed in section 7.2.6. However, this is not required, as the node also will change its IPv6 address. • Anycast neighbor solicitation is not supported. • Redirect messages are not supported (specification section 8). • Nodes which use multicast do not use the MLD protocol to announce groups they are members of.

IPv6 Stateless Address Autoconfiguration	
4862	• Maximum number of NS for DAD is limited to 1. • Variable *RETRANS_TIMER* is not configurable but fixed to value of 3s. • Address deprecation is not implemented. Behavior is the same as preferredLifetime==validLifetime. However, router advertisements with a perferredLifetime>validLifetime are ignored by the device.
Transmission of IPv6 Packets over IEEE 802.15.4 Networks	
4944	• ZMDIs implementation does not support one of the specified header compression algorithms proposed by the RFC. Instead, it implements the RFC draft-hui-6lowapn-hc version 01. 0x03 is used as dispatch value for this compression format. • Mesh functionality specified in the RFC is not implemented. Instead, a proprietary link-layer mesh implementation is provided. • Only 64-bit link-layer addresses are supported at this time.
IPv6 Configuration in Internet Key Exchange Protocol Version 2 (IKEv2)	
5996	• Supports only the negotiation of an ESP in transport mode between two protected endpoints. • The Diffie-Hellman group cannot be changed. • Only one initial exchange can occur at the same time. • Only one pair of child SAs can be negotiated with one IKE SA. • Windowing is not supported. • Timeouts are defined by user. • The critical flag is ignored. • Cookies are not supported. • Implementation provides only one proposal. • Only packets for port 500 are accepted. • EAP is not supported. • IPComp is not supported. • NAT traversal is not supported. • Only ID_IPV6_ADDR is supported. • Only shared key message integrity code are supported. • Vendor ID payload is not supported. • Configuration payload is not supported.
Cryptographic Algorithms for Use in the Internet Key Exchange Version 2 (IKEv2)	
4307	• Supports only 768 MODP Group. • Supports only ENCR_AES_CBC. • Supports only PRF_AES128_CBC. • Supports only AUTH_AES_XCBC_96.
6LoWPAN Compression of IPv6 Datagrams	
draft-hui-6lowpan-hc-01	• ISA100_UDP Header Compression is not implemented (specification section 3.3).

4 Core-Library Reference

4.1. Initialization

The core library provides two different hooks that can be used to initialize the application during system startup. The first hook, named **`ZWIR_AppInitHardware`**, is called before network initialization. The second one, named **`ZWIR_AppInitNetwork`**, is called afterwards.

```
void
  ZWIR_AppInitHardware ( ZWIR_ResetReason_t  resetReason )
```

This hook is called after power on and after reset to configure the module peripherals. The ***`resetReason`*** argument specifies the reset source that triggered the execution of this function. If required, the operating mode of the module *SHOULD* be set from this function.

Note: Most API functions must not be called from this function. The documentation specifies which API functions can be called from ZWIR_AppInitHardware.

```
void
  ZWIR_AppInitNetwork ( ZWIR_ResetReason_t  resetReason )
```

This hook is called when the default networking parameters have been initialized after power-on or reset. It should be used to open sockets and initialize further network parameters if required. However, it must not be used for sending out data, as the node has not completed Duplicate Address Detection (refer to section 3.8.4 for further details) at this point in time. The ***`resetReason`*** argument specifies the reset source that triggered the execution of this function.

```
void
  ZWIR_AppInitNetworkDone ( ZWIR_ResetReason_t  resetReason )
```

This hook is called after successful completion of the Duplicate Address Detection procedure. It is also called when DAD is disabled. In this case the function is called immediately after the call to **`ZWIR_AppInitNetwork`**. The ***`resetReason`*** argument specifies the reset source that triggered the execution of this function. All API functions may be called from this function.

```
void
  ZWIR_SetOperatingMode ( ZWIR_OperatingMode_t          opMode,
                          ZWIR_RadioReceiveCallabck_t   callback )
```

This function sets the operating mode of the device. Device Mode, Gateway Mode or Sniffer Mode may be selected (refer to section 3.5) with the ***`operatingMode`*** argument. The ***`callback`*** argument is used in Gateway Mode and Sniffer Mode to specify the function to be called on the reception of data. If callback is ***`NULL`***, no function will be called. If Device Mode is selected ***`callback`*** is ignored. It is *RECOMMENDED* to call **`ZWIR_SetOperatingMode`** from **`ZWIR_AppInitHardware`** only.

typedef enum { ... } ZWIR_OperatingMode_t

Type enumerating the different operating modes of the device. Possible values include:

ZWIR_omNormal = 0	Device Mode
ZWIR_omGateway = 1	Gateway Mode
ZWIR_omSniffer = 2	Sniffer Mode

typedef enum { ... } ZWIR_ResetReason_t

Type enumerating the different reasons for system reset. Possible values include:

ZWIR_rPowerOnReset	The device has been powered on after being switched off
ZWIR_rStandbyReset	The device is waking up from standby mode
ZWIR_rIndependentWatchdogReset	The MCU's independent watchdog (IWDG) was triggered
ZWIR_rSoftwareReset	A software reset has been performed
ZWIR_rPinReset	System reset was triggered by pulling low the reset pin
ZWIR_rWindowWatchdogReset	The window watchdog has triggered
ZWIR_rLowPowerReset	The supply voltage dropped below the specified threshold

4.2. Program Control

The API does not provide the concept of a central main function as usual in traditional C programs. Merely three timing driven hooks, named **ZWIR_Main10ms**, **ZWIR_Main100ms** and **ZWIR_Main1000ms**, are provided, which are called periodically after 10, 100 and 1000 milliseconds, respectively. Sensing and acting of the user application should be implemented in these hooks. For more fine-tuned time control, a user-programmable callback timer is available. This timer can be programmed at 1 millisecond increments. Initialization and deinitialization of the freely programmable timer function is accomplished by **ZWIR_StartCallbackTimer** and **ZWIR_StopCallbackTimer**.

The **ZWIR_Main10ms**, **ZWIR_Main100ms** and **ZWIR_Main1000ms** hooks can be defined to implement application behavior that has to be executed periodically. The default implementations of these hooks do nothing, so these hooks can be left undefined if they are not required.

Besides the fixed period main functions, the API also provides a freely configurable callback timer. The timer is started using the **ZWIR_StartCallbackTimer** function. It is possible to provide a data pointer to this function, which is passed to the callback when the timer expires. This allows for delayed data processing. It can be chosen, if the timer is triggered just once or periodically. The timer is stopped with the **ZWIR_StopCallbackTimer**.

```
void
  ZWIR_Reset ( void )
```

This function causes a software reset of the system. Both, the microcontroller and the transceiver are reset. The complete startup sequence is executed. If this function is called while the transceiver receives or transmits data, the packet will be lost.

```
void
  ZWIR_ResetNetwork ( void )
```

This function resets the radio transceiver and reinitializes the network stack. After a call to this function, IPv6 address auto configuration is restarted and manually assigned addresses are lost. Also routing and address resolution information are lost.

```
void
  ZWIR_Main10ms ( void )
void
  ZWIR_Main100ms ( void )
void
  ZWIR_Main1000ms ( void )
```

These hooks are called with a period of 10, 100 and 1000 milliseconds. On timeslots that are multiples of 10ms or 100ms, the shorter period function has priority over the longer period. This means **ZWIR_Main10ms** is called before **ZWIR_Main100ms,** which is called before **ZWIR_Main1000ms**. All three functions are called immediately at system startup.

Note: These functions are not suitable if exact timer behavior is required. A constant execution interval cannot be guaranteed, nor is it guaranteed that the function is executed at each intended time instant.

```
void
  ZWIR_StartCallbackTimer ( uint32_t                timeout,
                            ZWIR_TimeoutCallback_t  callback,
                            void*                   data,
                            bool                    periodic )
```

If this function is called, the freely programmable timer is initialized and started. The function provided with the ***callback*** argument will be called about ***timeout*** milliseconds after the call to **ZWIR_StartCallbackTimer**. The value provided with ***data*** will be passed to ***callback*** when it is called. If the ***periodic*** flag is set to one, ***callback*** is called periodically; otherwise ***callback*** is called just once. If this function is called while the timer is running, the timer will be reprogrammed and the previous programming will be lost.

Note: The callback timer is not suitable if exact timer behavior is required. Consider using a MCU timer peripheral for exact timing.

```
void
  ZWIR_StopCallbackTimer ( )
```

This function stops a running callback timer. If no timer is running, nothing will happen.

```
void
  ZWIR_TriggerAppEvent ( uint8_t  eventId )
```

This function allows the processing of application events with a certain operating system priority. This function notifies the operating system of the presence of an application event and schedules the appropriate callback function for execution. Typically this is used to execute computational intensive code in response to an interrupt.

```
void
  ZWIR_RegisterAppEventHandler ( uint8_t                 eventId,
                                 ZWIR_AppEventHandler_t  handler )
```

This function registers an application event handler callback for a certain application event in the operating system. If this function is called more than once with the same ***eventId***, the callback function provided with the last call will be in effect.

```
typedef  void ( * ZWIR_AppEventHandler_t ) ( void )
```

This function pointer type defines the signature of a callback function that is executed in response to an application event.

```
typedef  void ( * ZWIR_TimeoutCallback_t ) ( void*  data )
```

Function pointer type for the callback function that should be called if the callback timer expires.

```
ZWIR_RevisionInfo_t
  ZWIR_GetRevision ( void )
```

This function returns a structure containing detailed version information. This information must be provided if support requests are sent to ZMDI.

```
typedef struct { int8_t   majorRevision
                 int8_t   minorRevision
                 int16_t  versionExtension } ZWIR_RevisionInfo_t
```

Type for objects carrying version information. If problems are encountered while using the stack, request this structure using **ZWIR_GetRevision** and provide the information obtained to ZMDI together with an error report.

```
bool
  ZWIR_Error ( int32_t  errorCode )
```

This function is called when a recoverable library error is encountered. The error-code is passed in the ***errorCode*** argument. If *true* is returned, the error is assumed to be processed and no action will be taken by the stack. If *false* is returned, the default error handler will be executed.

```
int32_t
  ZWIR_Rand ( void )
```

This function returns a random number. The sequence of numbers generated by this function is actually only pseudo-random, as a linear feedback shift register with a generator polynomial is used. However, by calling **ZWIR_SRand** with zero as argument, the random number generator is seeded with a true random number.

```
int32_t
  ZWIR_SRand ( int32_t  seed )
```

This function seeds the random number generator. If zero is provided as ***seed***, a true random number is generated from thermal noise. Any value other than zero is used to initialize the generator directly. This allows creating reproducible application behavior for debug purposes. Using non-zero seed values in production code is not recommended.

4.3. Networking

A ZWIR451x node can join any IPv6 network. Each device automatically gets an IPv6 address that is computed from the link-layer address of the module. Additionally, the user's own IPv6 addresses can be assigned to the module. The modules can communicate bidirectional using the UDP protocol.

4.3.1. Address Management

The API provides a set of functions for managing the different addresses of a 6LoWPAN module. A module has three different types of addresses: the PAN identifier, the link-layer address, which is also called the PAN-address and a set of IPv6 addresses.

4.3.1.1. PAN Identifier

The PAN identifier is a 16-bit value that determines the personal area network that the module belongs to. All devices in a PAN must have the same PAN identifier. Devices with different PAN identifiers cannot communicate with each other. They cannot even use each other as a network relay. Each device has exactly one PAN identifier. It can be read and altered using the **ZWIR_SetPANId** and **ZWIR_GetPANId** functions, respectively. The default value of the PAN identifier is $ACCA_{HEX}$.

```
uint16_t
  ZWIR_GetPANId ( void );
```

Reads and returns the PAN identifier of the module.

```
void
  ZWIR_SetPANId ( uint16_t  panId );
```

Sets the PAN identifier of the node to the value provided in ***panId***. The value is retained until the next reset or until Standby Mode is entered. After waking up, the factory programmed value is active again until the next call to this function.

4.3.1.2. Link-Layer Address

The link-layer address, also called PAN-address, is a 64-bit-wide value that identifies the device in the PAN. All communication between devices in a PAN is based on the link-layer address. Higher layer protocols, such as IPv6, resolve their addresses into link-layer addresses. A globally unique link-layer address is programmed into each device during manufacturing. Changing this address is not recommended, as this could cause the address to be not longer unique. Nevertheless, the functions **`ZWIR_SetPANAddress`** and **`ZWIR_GetPANAddress`** allow reading and writing the PAN-address.

```
ZWIR_PANAddress_t const*
  ZWIR_GetPANAddress ( void )
```

Reads and returns the link-layer address of the module.

```
void
  ZWIR_SetPANAddress ( ZWIR_PANAddress_t const*  panAddress )
```

Sets the link-layer address to the value provided in ***`panAddress`***. Changing the link-layer address of module is not recommended for those reasons given in section 3.8.4. However, if manual assignment of addresses is required, calling **`ZWIR_SetPANAddress`** from **`ZWIR_AppInitHardware`** is recommended. The assigned ***`panAddress`*** value is retained until the next system reset or deep sleep. After waking up, the factory programmed value is active again until the next call to this function.

Note: Changing the link-layer address of a device during normal operation will typically cause a loss of all incoming packets for a period of time. The standard allows sending unsolicited neighbor advertisements as an option if the link-layer address changes. This feature is not included in ZMDI's 6LoWPAN stack.

```
typedef union { uint8_t   u8  [ 8 ],
                uint16_t  u16 [ 4 ],
                uint32_t  u32 [ 2 ] } ZWIR_PANAddress_t
```

Data type for representation of link-layer addresses. Bytes are stored in network byte order, which is big endian. That means that the highest order byte is stored first; therefore using the u16 or u32 elements might cause unexpected results especially when printing. Consider using the function **`ntohs`** for u16 and **`ntohl`** for u32 elements if host byte order is required.

4.3.1.3. IPv6 Addresses

Although manual assignment of IPv6 addresses is not required by most applications, the API provides the function **`ZWIR_SetIPv6Address`**. This function can be used to assign additional IPv6 addresses to an interface. Up to three addresses can be assigned in total, but the first address is always allocated by the automatically configured link-local address. The function **`ZWIR_GetIPv6Addresses`** can be used to request all addresses assigned to an interface.

The **`ZWIR_CheckMulticastGroup`** hook is called if a multicast packet is received that does not belong to the all nodes multicast group or the node solicited address multicast group. This function must be implemented by the application if multicast addressing is used.

```
bool
  ZWIR_SetIPv6Address ( ZWIR_IPv6Address_t const*  ipv6 )
```

Add the address provided in ***ipv6*** to the network interface.

The function returns *true* if the operation was successful or *false* otherwise. The function fails if the maximum number of IPv6 addresses is assigned to the interface already.

```
uint8_t
  ZWIR_GetIPv6Addresses ( ZWIR_IPv6Address_t const*  ipv6Buffer,
                          uint8_t                    maxCount )
```

Request a set of addresses assigned to the interface. The ***ipv6Buffer*** argument must carry a pointer that points to a buffer that is able to store at most ***maxCount*** IPv6Addresses. The function will store ***maxCount*** addresses in this buffer if the interface has at least ***maxCount*** addresses assigned. If the number of assigned address is lower than ***maxCount***, the function will store all available addresses.

The return value determines the number of addresses that have actually been stored. Thus, if 0 is returned, the interface has no IPv6 address assigned. This could be due to failing duplicate address detection.

```
void
  ZWIR_SetDestinationPANId ( uint16_t  pandID )
```

This function is used for changing the destination PAN Id temporarily. The configured value remains in effect until **ZWIR_ResetDestinationPANId** is called or the device is reset.

```
void
  ZWIR_ResetDestinationPANId ( void )
```

This function reset the PAN Id of the device to the last value that had been configured before it was changed using **ZWIR_SetDestinationPANId**.

```
bool
  ZWIR_CheckMulticastGroup ( ZWIR_IPv6Address_t const*  ipv6 )
```

This hook is called whenever a multicast packet is received that contains a multicast group that is not known by the network stack. The user implementation must decide if the node is part of the multicast group provided with the multicast group ID (the lower 112 bytes) in the IPv6 address. *True* must be returned if the node belongs to the multicast group, *false* otherwise.

```
typedef union { uint8_t   u8  [ 16 ],
                uint16_t  u16 [  8 ],
                uint32_t  u32 [  4 ] } ZWIR_IPv6Address_t
```

Data type for representation of IPv6 addresses. Bytes are stored in network byte order, which is big endian; i.e., the highest order byte is stored first. Therefore using the u16 or u32 elements might cause unexpected results especially when printing. Consider using the functions **ntohs** for u16 and **ntohl** for u32 elements if host byte order is required.

4.3.2. Socket and Datagram Handling

The functions **ZWIR_OpenSocket** and **ZWIR_CloseSocket** are provided for opening and closing sockets, respectively. Datagrams are sent over sockets using the **ZWIR_SendUDP**, **ZWIR_SendUDP2** and **ZWIR_Send6LoWPAN** functions, depending on the operating mode of the device.

Incoming datagrams are handled by user-defined callback functions, which must be to be assigned to sockets using the **ZWIR_OpenSocket** function. The API functions **ZWIR_GetPacketSenderAddress**, **ZWIR_GetPacketSenderPort** and **ZWIR_GetPacketHopCount** are provided for requesting the address and port of a sender.

```
ZWIR_SocketHandle_t
  ZWIR_OpenSocket ( ZWIR_IPv6Address_t const*     remoteAddr
                    uint16_t                      remotePort,
                    uint16_t                      localPort,
                    ZWIR_RadioReceiveCallback_t   rxHandler )
```

This function opens a new socket to a remote host. The ***remoteAddr*** and ***remotePort*** arguments specify the IPv6 address and the UDP port of the remote host. The ***localPort*** argument specifies the port on which incoming data is accepted. If ***localPort*** is set to zero, an unused port from the range from 4096 through 32000 is chosen. If incoming data should be received, a pointer to a callback function must be passed in the ***rxHandler*** argument. If no data is expected, the value of ***localPort*** does not matter and ***rxHandler*** must be set to NULL. In order to receive packets from arbitrary remote hosts, the unspecified address can be passed to the function. In this case, the socket is not suitable for sending packets.

On success, the function returns a socket handle that can be used with datagram handling functions. The function fails if the maximum number of sockets is already opened or if a socket with the same parameters ***remoteAddress*** and ***localPort*** already exists. In this case NULL is returned.

```
void
  ZWIR_CloseSocket ( ZWIR_SocketHandle_t  socket )
```

Open sockets are closed using this function. If a **socket** is invalid or has already been closed, the function has no effect. Closing a socket has no effect on any previously sent packets, even if transmission is not yet completed.

```
bool
  ZWIR_SendUDP ( ZWIR_SocketHandle_t    socket,
                 uint8_t const*         data,
                 uint16_t               length )
```

UDP datagrams are sent over a specific socket using this function. The socket denoted by the ***socket*** argument determines the destination address and port of the datagram. The ***data*** and ***length*** arguments specify the payload. The maximum packet size is 1232 byte. The function returns a non-zero value if the packet to be sent was successfully queued in the output queue, otherwise zero is returned. If zero is returned, there is not enough room in the output queue to buffer this packet. In this case control must be passed back to the operating system.

Note: A non-zero return value does not automatically denote the successful delivery of the packet. Successful delivery can only be verified by response-packets sent on the application level.

Note: Calling this function in a while loop, waiting for a non-zero result will dead-lock the system if a packet cannot be queued. After a zero result control always must be passed to the operating system. Otherwise the output buffers will never be freed and this function continues to fail, resulting in a dead-lock.

Note: This function cannot be used in the Gateway Mode; use **ZWIR_Send6LoWPAN** when in Gateway Mode!

```
bool
  ZWIR_SendUDP2 ( uint8_t*              data,
                  uint16_t              size,
                  ZWIR_IPv6Address_t*   remoteAddress,
                  uint16_t              remotePort )
```

This function sends an UDP packet without the need for opening a socket. Destination address and destination port are provided in the ***remoteAddress*** and ***remotePort*** arguments. The local UDP port is selected arbitrarily by the network stack. The maximum packet size is 1232 byte. The function returns a non-zero value if the packet to be sent was successfully queued in the output queue, otherwise zero is returned. If zero is returned, there is not enough room in the output queue to buffer this packet. In this case control must be passed back to the operating system.

Note: A non-zero return value does not automatically denote the successful delivery of the packet. Successful delivery can only be verified by response-packets sent on the application layer.

Note: Calling this function in a while loop, waiting for a non-zero result will dead-lock the system if a packet cannot be queued. After a zero result control always must be passed to the operating system. Otherwise the output buffers will never be freed and this function continues to fail, resulting in a dead-lock.

Note: This function cannot be used in the Gateway Mode; use **ZWIR_Send6LoWPAN** when in Gateway Mode!

```
ZWIR_IPv6Address_t const*
  ZWIR_GetPacketSenderAddress ( void )
```

This function returns a pointer to the IPv6 address of the last received packet. It can be used reliably in the RX callback function. Using this function outside of the RX callback function might cause unpredictable results.

Note: This function cannot be used in Gateway Mode.

```
uint16_t
  ZWIR_GetPacketSenderPort ( void )
```

This function returns the sender port of the last received packet. It can be used reliably in the RX callback function. Using this function outside of the RX callback function might cause unreliable results.

Note: This function cannot be used in Gateway Mode.

```
uint8_t
  ZWIR_GetPacketHopCount ( void )
```

Returns the number of hops the last received packet has taken.

Note: Using this function outside of the RX callback function might cause unreliable results.

```
int32_t
  ZWIR_GetLastRSSI ( void )
```

Returns the receive signal strength indicator (RSSI). The value approximately corresponds to the receive power level in dBm. Using this function outside of the RX callback function might cause unreliable results.

```
ZWIR_PANAddress_t*
  ZWIR_GetSourcePANAddress ( void )
```

Returns the source PAN address of the latest received packet.

Note: Using this function outside the receive callback might cause unreliable results.

```
ZWIR_PANAddress_t*
  ZWIR_GetDestinationPANAddress ( void )
```

Returns the destination PAN address of the last received packet.

Note: Using this function outside the receive callback might cause unreliable results.

```
ZWIR_SocketHandle_t
  ZWIR_GetPacketRXSocket ( void )
```

Returns the socket handle of the socket the last packet was received on.

```
ZWIR_IPv6Address_t*
  ZWIR_GetFailingAddress ( void )
```

Retruns the last address address resolution failed for. This function is typically called from **ZWIR_Error** when the **ZWIR_eHostUnrechable** error was reported. When no address resolution error occurred since the last reset, the result is undefined.

```
bool
  ZWIR_Send6LoWPAN ( ZWIR_PANAddress_t const*  remoteAddr,
                     uint16_t const*           data,
                     uint8_t const             length )
```

This function is used to send complete IPv6/UDP packets to the remote host with the link-local address ***remoteAddr***. No UDP or IPv6 header processing is performed on the packet. Instead it is passed directly to the 6LoWPAN processing layer. The ***data*** argument must point to the first header byte of the IPv6/UDP packet; ***length*** specifies the size including all headers. This is useful in conjunction with the Gateway Mode.

Note: Calling this function in a while loop, waiting for a non-zero result will dead-lock the system if a packet cannot be queued. After a zero result control always must be passed to the operating system. Otherwise the output buffers will never be freed and this function continues to fail, resulting in a dead-lock.

```
typedef
   void ( * ZWIR_RadioReceiveCallback_t ) ( uint8_t*  data,
                                            uint16_t  length )
```

Function pointer type for the callback function that should be called on reception of data over an UDP socket.

```
typedef
  void*  ZWIR_SocketHandle_t
```

Type representing a socket.

4.3.3. Radio Parameters

The radio module provides the capability to alter the physical radio channel and the transmit output power. This is accomplished using the **ZWIR_SetChannel**, **ZWIR_SetModulation** and **ZWIR_SetTransmitPower** functions. Changes to the radio parameters are getting into effect immediately. The changes are reset by ZWIR_Reset, but not by ZWIR_ResetNetwork. Changing radio parameters during the transmission/reception of a packet will very likely cause the loss of the packet.

```
void
  ZWIR_SetChannel ( ZWIR_RadioChannel_t  channel )
```

Sets the module to the radio channel specified by ***channel***.

Note: If this is done while a transmission or reception is ongoing, the transmitted or received packet will be lost. It is recommended to call this function from **ZWIR_AppInitHardware**.

Note: Local regulations limit the use of the spectrum. You *MUST* only select channels which are allowed to be used in the area the application is going to be installed! Check with you local authorities which part of the spectrum is allowed to be used for your application.

```
void
  ZWIR_SetModulation ( ZWIR_Modulation_t  modulation )
```

This function is used to change the modulation scheme to the value specified with the ***modulation*** argument.

Note: If this is done while a transmission or reception is ongoing, the transmitted or received packet will be lost. It is recommended to call this function from **ZWIR_AppInitHardware**.

```
void
  ZWIR_SetTransmitPower ( int  power )
```

Sets the transceiver output power to the value specified by ***power***. The valid range of values depends on the channel being selected. In the European frequency band a transmission power of -10dBm to 5dBm may be selected; in the US band -10dBm to 10dBm are possible. Values being too low or too high are automatically adjusted to the closest valid value.

Note: If this is done while a transmission is ongoing the transmitted packet is very likely to be lost. For that reason it is recommended that this function be called only from **ZWIR_AppInitHardware**.

```
typedef enum { ... } ZWIR_RadioChannel_t
```

Radio channel enumeration type accepted by **ZWIR_SetChannel**. Possible values include

ZWIR_channel0, ZWIR_eu868	EU-Band, 868.3 MHz
ZWIR_channel1, ZWIR_us906	US-Band, 906 MHz
ZWIR_channel2, ZWIR_us908	US-Band, 908 MHz
ZWIR_channel3, ZWIR_us910	US-Band, 910 MHz
ZWIR_channel4, ZWIR_us912	US-Band, 912 MHz
ZWIR_channel5, ZWIR_us914	US-Band, 914 MHz
ZWIR_channel6, ZWIR_us916	US-Band, 916 MHz
ZWIR_channel7, ZWIR_us918	US-Band, 918 MHz
ZWIR_channel8, ZWIR_us920	US-Band, 920 MHz
ZWIR_channel9, ZWIR_us922	US-Band, 922 MHz
ZWIR_channel10, ZWIR_us924	US-Band, 924 MHz
ZWIR_channel100,ZWIR_eu865	EU-Band, 865.3 MHz
ZWIR_channel101,ZWIR_eu866	EU-Band, 866.3 MHz
ZWIR_channel102,ZWIR_eu867	EU-Band, 867.3 MHz

```
typedef enum { ... } ZWIR_Modulation_t
```

Enumeration of modulation schemes accepted by **ZWIR_SetModulation**. Possible Values include

ZWIR_mBPSK	Binary Phase Shift Keying
ZWIR_mQPSK	Offset Quadrature Phase Shift Keying

4.3.4. Gateway Mode Functions

The ZWIR45xx network stack provides the Gateway Mode to allow easy implementation of network bridges. Therefore, the stack does only perform network processing up to and including the 6LoWPAN layer. Thus, when a packet comes in, a full IPv6 packet is passed to the receive callback. This allows the implementation of a perfectly transparent network bridge. However, if the bridge should be updateable or configuration parameters should be set remotely, it would be desirable to have the opportunity of performing higher layer processing of incoming packets which are addressed to the bridge. For this purpose the types and functions defined in this section are provided.

```
bool
  ZWIR_GatewayProcessPacket ( uint8_t*   data,
                              uint16_t   size );
```

This function must be called to make the network stack process the network and higher layer protocols of an incoming packet. The function is intended to be called from the receive callback of gateway mode devices. The ***data*** and ***size*** arguments of the receive callback should be passed unmodified to this function.

```
void
  ZWIR_GatewaySetOutputFunction ( ZWIR_GatewayOutputFunction_t   fn )
```

Devices operating in gateway mode typically have more than one network interface. When higher layer protocol processing is in place, it must be decided which network interface is used for sending out packets. This decision has to be taken by an application callback which must be registered at the network stack using this function.

```
typedef
  uint8_t ( *ZWIR_GatewayOutputFunction_t ) ( uint8_t*             data,
                                              uint16_t             size,
                                              ZWIR_PANAddress_t*   address );
```

This type defines the signature of functions to be used as output function in gateway mode. The task of this function is taking the decision to which interface an outgoing packet must be sent and calling the corresponding output function for this interface. The decision is typically taken based on the PAN address of the destination node which is provided in the ***address*** argument. The ***data*** argument carries a pointer to the IPv6 packet to be sent; the ***size*** argument determines the size of this packet.

4.3.5. Miscellaneous

```
ZWIR_TRXStatistic_t
  ZWIR_GetTRXStatistic ( void )
```

This function returns statistic information about transmission and reception. The returned data structure contains the number of packets, the number of bytes received and transmitted, the number of retransmissions and the number of CRC failures on reception. Furthermore, the transmit duty-cycle is included. The counters are reset either on reset or if **`ZWIR_ResetTRXStatistic`** is called.

Checking the duty cycle should be performed on a regular basis in order to meet the duty cycle requirements at the operation site of the device. Please contact the local authorities to find out if duty-cycle limitations apply on your target market(s).

Note: All values in the structure might be higher than expected. This is due to the overhead communication that is required for address resolution and route discovery. Refer to section 3.10 to check if it is possible to optimize constant settings in order to reduce overhead traffic to a minimum.

```
void
  ZWIR_ResetTRXStatistic ( void )
```

This function resets all values of the transceiver statistics to 0. This function has no effect on ongoing transfers.

```
typedef struct {
    uint32_t  txBytes
    uint32_t  txPackets
    uint32_t  rxBytes
    uint32_t  rxPackets
    uint32_t  txFail
    uint32_t  dutyCycle
} ZWIR_TRXStatistic_t
```

This structure is returned by **`ZWIR_GetTRXStatistic`**. The values contained are counted starting from reset, network reset (initiated by **`ZWIR_RestNetwork`**) or a call to **`ZWIR_ResetTRXStatistic`**. Besides data sent from the application code, the fields contained in the structure also consider packets that are sent in the background (e.g., route and neighbor discovery). The **`ZWIR_dutyCycle`** filed contains the quotient of time spent sending and the time elapsed since the occurrence of one of the above events. In order to obtain the actual duty cycle percentage divide **`dutyCycle`** by 1000.

```
void
  ZWIR_SetPromiscuousMode ( bool  enable )
```

This command puts the device into promiscuous reception mode. This means that on MAC layer all packet filtering is disabled. In promiscuous mode the device receives packets sent to all PAN Ids and all PAN Addresses, regardless of its own PAN Id and PAN Address configuration. Filters on higher protocol layers are still active.

The promiscuous mode should not be used in normal operation. It may make sense in Gateway Mode and it is required for sniffers.

```
bool
  ZWIR_CreateAlternativeAddressList ( uint16_t  size )
```

This function is used in promiscuous mode to allocate memory for an alternative PAN address list. The device will treat each packet sent to one of the addresses in the alternative PAN address list in the same way as if the packet had been sent to the device's own PAN address. The ***size*** argument determines the maximum number of entries in the list. The function returns **true** on success or **false** otherwise

```
bool
  ZWIR_AddAlternativeAddress ( ZWIR_PANAddress_t*              address,
                               ZWIR_AlternativeAddressType_t   type )
```

This function adds a PAN address to the alternative address list. The ***address*** argument specifies the address to be added and the ***type*** argument specifies the type of the address. The function returns **true** if the address was added successfully. **false** is returned when no alternative address list has been allocated (refer to **ZWIR_CreateAlternativeAddressList**). If ***address*** is already in the address list **true** is returned. If there is no free item in the alternative address list, the item that has not been used for the longest time is overwritten.

```
ZWIR_AlternativeAddressType_t
  ZWIR_IsAlternativeAddress ( ZWIR_PANAddress_t*              address,
                              ZWIR_AlternativeAddressType_t   type )
```

This function checks whether ***address*** of ***type*** is in the alternative PAN address list or not. ***type*** is used as filter. The type of address stored in the address list logically AND'ed with ***type.*** The function returns the type of the stored address if available or **ZWIR_noAddr** otherwise.

```
typedef  enum {...} ZWIR_AlternativeAddressType_t
```

This type is used by **ZWIR_IsAlternativeAddresss** and **ZWIR_AddAlterantiveAddress** as address type, address filter or return value. Possible values are:

ZWIR_aatNone	0x00	Address not found (only with **ZWIR_IsAlternativeAddress**)
ZWIR_aatEUI64	0x01	Address is a EUI64 address
ZWIR_aatEUI48	0x02	Address is a EUI48 address
ZWIR_aatAny	0x03	Only to be used as filter in **ZWIR_IsAlternativeAddress**

```
bool
  ZWIR_ExternalClockEnable ( bool  enable )
```

This function is used to select whether the external or the internal clock is used as system clock. The external clock is much more precise, but it is not possible to use sleep mode or turn off the transceiver when the external clock is used.

```
char const*
  ZWIR_GetFCCID ( void )
```

This function returns the FCC ID of the module. The FCC ID is returned as NULL-terminated string.

4.4. Power Management

The MCU on the module can operate at different clock rates. The lowest possible clock frequency matching the needs of the application should be selected in order to work as power efficiently as possible. The operating system frequency is set using **ZWIR_SetFrequency**.

Besides clock speed modification, the radio module supports the Sleep, Stop and Standby Modes (see section 3.6.3). The wakeup conditions are adjustable for each power mode individually. By default, the system continues its execution after an RTC alarm. The state of the transceiver depends of the selection of the transceiver interrupt or event as the wakeup source. When the transceiver interrupt or event is masked, the transceiver will be switched of automatically.

All three power modes can be executed immediately or delayed to send out all buffered packets. After entering the Standby Mode, all RAM content is lost and the microcontroller will be reset.

The functions **ZWIR_SetWakeupSource**, **ZWIR_PowerDown** and **ZWIR_AbortPowerDown** are used to configure, to enter or to leave the low power modes.

```
void
  ZWIR_SetFrequency ( ZWIR_MCUFrequency_t  freq )
```

This function sets the clock speed of the MCU core. Peripheral clocks are not changed.

```
void
  ZWIR_AbortPowerDown ( void )
```

This function stops all delayed power down actions.

```
void
  ZWIR_PowerDown ( ZWIR_PowerDownState_t  powerDownMode,
                   uint32_t               time )
```

This function changes the power mode of the system immediately or after sending all buffered fragments. The ***powerDownMode*** argument defines the next power down mode. The ***time*** parameter specifies the power down time. For the Sleep Mode, the time is given in 10 milliseconds multiples. For the Stop and Standby Modes, the time is given in seconds. If the RTC alarm is not selected as the source, the time parameter will be ignored.

```
void
  ZWIR_SetWakeupSource ( ZWIR_PowerDownState_t  powerDownMode,
                         uint64_t               wakeupSource )
```

This function sets the wakeup condition for a power mode. The ***powerDownMode*** argument defines the power down mode to be configured and the ***wakeupSource*** parameter specifies the event(s) that will cause the module to enter active mode, again. Depending on the value of ***powerDownMode***, the ***wakeupSource*** parameter is interpreted differently. The settings being applied to different register will be revoked when exiting power down mode.

In sleep mode each interrupt can be used to wake up the system. Accordingly, the ***wakeupSource*** parameter is interpreted as an interrupt mask. The interrupt mask allows selecting one or more of the lower 64 interrupts to be selected as wakeup source. The bits correspond to the interrupt position according to the Nested Vectored Interrupt Controller (NVIC) documentation in the STM32 Reference Manual.

For Stop Mode, only events are supported as wakeup source. Therefore, the ***wakeupSource*** parameter is used to configure the external interrupt/event controller's event mask register (EXTI_EMR). The external interrupt controller limits the wakeup sources for stop mode to the external pins, the programmable voltage detector, the real-time clock and the USB wakeup function. Refer to the STM32 Reference Manual for further information about the external interrupt controller.

Exit from standby mode is only possible using the Real-Time Clock (RTC) or the external wakeup pin. Wakeup by RTC is selected if '1' is passed as ***wakeupSource***, the WKUP pin is selected by '2' and an argument of '3' selects both.

If an invalid wakeup source is selected, the default wakeup source, the RTC, is set.

```
void
  ZWIR_Sleep ( uint16_t  sleepTime )
```

This function puts the system into Sleep Mode. The ***sleepTime*** argument controls the duration of Sleep Mode and is given in 10 millisecond multiples. That means that a ***sleepTime*** value of 100 puts the system into Sleep Mode for 1 second. During Sleep Mode, all memory contents are retained. Waking up the system from sleep is not possible.

Note: This function is deprecated – use **ZWIR_PowerDown** instead.

```
void
  ZWIR_Standby ( uint32_t  standbyTime )
```

This function puts the system into Standby Mode. The ***standbyTime*** argument controls the duration of Standby Mode and is given in seconds. In order to consume minimal power, almost all power domains of the MCU are disconnected. Memory contents are not retained during Standby Mode. The system can be woken up before expiration of the standby timer if the external wakeup pin of the module is triggered.

Note: This function is deprecated – use **ZWIR_PowerDown(*ZWIR_pStandby, standbyTime*)** instead

void
ZWIR_TransceiverOff (void)

This function switches the transceiver off manually. Attempts to send data using one of the functions **ZWIR_SendUDP**, **ZWIR_SendUDP2** and **ZWIR_Send6LoWPAN** while the transceiver is switched off will fail and the sent data will be lost. To turn the transceiver on, the functions **ZWIR_TransceiverOn** or **ZWIR_ResetNetwork** can be used. The transceiver is re-enabled after a system reset.

void
ZWIR_TransceiverOn (void)

This function switches the radio transceiver on manually after having it switched of using **ZWIR_TransceiverOff**. If the transceiver is already active, this function does nothing.

Typedef
enum { ... } ZWIR_PowerDownState_t

MCU frequency enumeration type accepted by **ZWIR_PowerDown** and **ZWIR_SetWakeupSource**. Possible values include

Value	Power Mode	Wait Until Sent?	Possible Wakeup Sources
ZWIR_pSleep	Sleep	No	IRQ 0 to 63
ZWIR_pSleepAfterActivities	Sleep	Yes	IRQ 0 to 63
ZWIR_pStop	Stop	No	EXTI 0 to 18
ZWIR_pStopAfterActivities	Stop	Yes	EXTI 0 to 18
ZWIR_pStandby	Standby	No	RTC, Wakeup pin
ZWIR_pStandbyAfterActivities	Standby	Yes	RTC, Wakeup pin

Typedef
enum { ... } ZWIR_MCUFrequency_t

MCU frequency enumeration type accepted by **ZWIR_SetFrequency**. Possible values include

ZWIR_mcu8MHz 8 MHz

ZWIR_mcu16MHz 16 MHz

ZWIR_mcu32MHz 32 MHz

ZWIR_mcu64MHz 64 MHz

4.5. Network Monitoring

ZWIR451x devices implement a protocol providing the functionality for remote network monitoring without the need for explicit support from the application layer. The monitoring functionality is available on each node operating in Device or Gateway Mode regardless of the application running on the module. Monitoring functions implemented at the time of publication of this document allow for discovery of network devices, determination of node configuration parameters and determination of routes through the network. The protocol is implemented on top of UDP.

```
void
  ZWIR_DiscoverNetwork ( ZWIR_DiscoveryCallback_t  callback,
                         uint8_t                   responseInterval )
```

This function initiates network discovery. A network discovery request is broadcasted to all nodes in the network. The ***callback*** argument is a pointer to a function that should be called to pass replying node information to the application. The information provided includes the hop-distance, the RSSI, IPv6 address count and all IPv6 addresses of the node. The ***responseInteval*** argument specifies a maximum time interval within which a responding device has to answer the request. The actual response time is chosen randomly. ***responseInterval*** should be increased with growing number of devices in the network. If zero is specified as ***responseInterval***, the default response time of three seconds is used.

Note: This function is deprecated! Use **ZWIR_NetMA_RemoteParameterRequest** instead.

```
void
  ZWIR_NetMA_RemoteParameterRequest ( ZWIR_IPv6Address_t*       address,
                                      ZWIR_NetMA_RPRCallback_t  callback,
                                      ZWIR_NetMA_RPRFields_t    fields = 0,
                                      ZWIR_NetMA_Flags_t        flags = 0,
                                      uint8_t                   respInterval = 3,
                                      uint8_t                   queryId = 0,
                                      uint8_t                   hopLimit = 0 )
```

Use this function to obtain configuration data remotely. Data may be requested from a single or from multiple devices. Different parameters control the answering of the requested devices. The function is actually a macro providing the possibility of using the function like a C++ style function with default arguments.

The ***address*** parameter specifies the device(s) from which data is requested. The function provided with ***callback*** will be called when a response is received. The remaining arguments are optional. As in C++ all arguments before the last one to be specified need to be provided. Assuming the application needs to specify the ***respInterval*** argument, ***fields*** and ***flags*** have to be specified as well.

The ***fields*** argument controls which sets of information are requested from the remote device. Refer to the documentation of the **ZWIR_NetMA_RPRFields_t** to see which options are available. ***flags*** limits the scope of the request which is especially useful in conjunction with multicast addressing. For instance it is possible to send requests just to devices configured in Gateway Mode using the ***flags*** argument. ***respInterval*** specifies the maximum number of seconds a device waits before it sends its response. The actual response interval is chosen randomly between zero and ***respInterval*** seconds in order to avoid collisions of responses sent from multiple devices at the same time.

The ***queryId*** is used to distinguish between different queries. A device that has successfully responded to a remote parameter request will not respond to another remote parameter request with the same ***queryId***. However, the ***queryId*** is only considered when the corresponding flag in the ***flags*** argument is set.

The ***hopLimit*** argument specifies up to which hop limit devices may respond to the request. If ***hopLimit*** is left unspecified or explicitly set to zero, all devices will respond regardless of their hop-distance to the requesting device. However, the ***hopLimit*** is only considered when the corresponding flag in the ***flags*** argument is set.

```
typedef
  void ( *ZWIR_NetMA_RPRCallback_t ) ( ZWIR_NetMA_RPRFields_t     fields,
                                       ZWIR_NetMA_RemoteData_t*   data )
```

This type defines the signature of functions to be used as remote parameter request response callbacks.

```
void
  ZWIR_NetMA_SetPort ( uint16_t  port )
```

Using this function the application may change the ***port*** used by the NetMA protocol. The default UDP port is 1357.

```
void
  ZWIR_NetMA_Trace ( ZWIR_PANAddress_t*          routeDestination,
                     ZWIR_IPv6Address_t*         routeSource,
                     ZWIR_NetMA_TraceCallback_t  callback )
```

This function allows examining routes through the network. Routes to ***routeDestination*** may be examined from the node calling this function or from a starting point which is passed as in the ***routeSource*** argument. Selecting a different starting point is required for requests coming from nodes not implementing ZMDIs network stack, e.g. computers in a network.

Responses to route requests are received by the application through the function defined by callback. If callback is NULL, the trace request will not be executed. Replied route information contains the list of all hops to ***routeDestination*** along with the forward RSSI of each hop. Be aware that response times may be significant long if long routes are examined.

Note: This function will not create routes between ***routeSource*** and ***routeDestination***, but may generate routes between the requesting device and ***routeSource*** if required.

```
typedef
  void ( * ZWIR_DiscoveryCallback_t ) ( uint8_t              hopCount,
                                        int8_t               rssi,
                                        uint8_t              addrCount,
                                        ZWIR_IPv6Address_t*  addresses )
```

Function pointer type for the callback function that should be called if network discovery reply packets are received.

```
typedef
  void ( * ZWIR_NetMA_TraceCallback_t ) ( uint8_t                 hopCount,
                                          ZWIR_NetMA_HopInfo_t*   hopInfo )
```

This type defines the function signature that is required by functions to be used as callback for the ZWIR_NetMA_Trace function.

```
typedef
  struct {
    ZWIR_PANAddress_t  address;
    int16_t            linkRSSI;
  } ZWIR_NetMA_HopInfo_t
```

Objects of this type are passed to the trace route callback function. Each object contains the address of a hop and the RSSI value of the forward path from the previous hop/source node to this node. Note that the return path's RSSI might be slightly different.

```
typedef
  enum { ... } ZWIR_NetMA_RPRFields_t
```

This enumerator is used with **ZWIR_NetMA_RemoteParameterRequest** to specify which sets of information have to be included in the response. The values may be binary OR'ed to request multiple sets of information at the same time. The following values are available:

ZWIR_NetMA_rprfMACAddress	0x0100	include **ZWIR_NetMA_RemoteMACAddr_t**
ZWIR_NetMA_rprfFirmwareVersion	0x0200	include **ZWIR_NetMA_RemoteVersion_t**
ZWIR_NetMA_rprfConfig	0x0400	include **ZWIR_NetMA_RemoteConfig_t**
ZWIR_NetMA_rprfIPv6Addresses	0x0800	include **ZWIR_NetMA_RemoteIPv6Addr_t**
ZWIR_NetMA_rprfTRXStatistics	0x1000	include **ZWIR_NetMA_RemoteStatus_t**

```
typedef
  enum { ... } ZWIR_NetMA_Flags_t
```

This enumerator defines flags to be used in conjunction with remote parameter requests. The flags limit the scope of the request. Possible values include:

ZWIR_NetMA_fDevice	0x04
ZWIR_NetMA_fBridge	0x08
ZWIR_NetMA_fQueryID	0x10
ZWIR_NetMA_fHopCountLimitation	0x20

```
typedef
  struct {
    ZWIR_NetMA_RemoteMACAddr_t*    macAddr;
    ZWIR_NetMA_RemoteIPv6Addr_t*   ipv6Addr;
    ZWIR_NetMA_RemoteConfig_t*     config;
    ZWIR_NetMA_RemoteVersion_t*    version;
    ZWIR_NetMA_RemoteStatus_t*     status;
  } ZWIR_NetMA_RemoteData_t
```

This structure contains pointers to the remote parameters received in response to a remote parameter request. The fields correlate with the **`ZWIR_NetMA_RPRFields_t`** enumerators. For each requested field the corresponding pointer should be set in the response. Fields which have not been requested result in a NULL pointer of the corresponding structure element.

```
typedef
  struct {
    uint16_t           panID;
    ZWIR_PANAddress_t  panAddr;
  } ZWIR_NetMA_RemoteMACAddr_t
```

This structure type carries a remote device's configured PAN ID and its PAN address.

```
typedef
  struct {
    uint8_t              count;
    ZWIR_IPv6Address_t   addresses [ ];
  } ZWIR_NetMA_RemoteIPv6Addr_t
```

This type defines a structure carrying all IPv6 addresses of a device. The count argument defines how many addresses are contained in the structure; the addresses element contains the actual addresses. The size of memory required by this structure varies – it depends on the number of contained addresses. The size of this structure cannot be determined using C's **`sizeof`** operator.

```
typedef
  struct {
    uint16_t  routeTimeout;
    uint16_t  routingTableSize;
    uint16_t  neighborReachableTime;
    uint8_t   neighborCacheSize;
    uint8_t   maxNetfloodHopCount;
    uint8_t   maxSocketCount;
    uint8_t   routeMaxFailCount;
    int8_t    routeRequestMinLinkRSSI;
    uint8_t   routeRequestMinLinkRSSIReduction;
    uint8_t   routeRequestAttempts;
    uint8_t   channel;
    uint8_t   power;
    uint8_t   modulation;
    uint8_t   doDuplicateAddressDetection;
    uint8_t   doRouterSolicitation;
  } ZWIR_NetMA_RemoteConfig_t
```

Objects of this type are used to report the configuration of the remote device.

```
typedef
  ZWIR_TRXStatistic_t  ZWIR_NetMA_RemoteStatus_t
```

This type defines the remote status as being equal to the transceiver statistics type.

```
typedef
  struct {
    uint32_t  vendorID;
    uint16_t  productID;
    uint8_t   firmwareMajorVersion;
    uint8_t   firmwareMinorVersion;
    uint16_t  firmwareVersionExtension;
    uint8_t   libraryMajorVersion;
    uint8_t   libraryMinorVersion;
    uint16_t  libraryVersionExtension;
  } ZWIR_NetMA_RemoteVersion_t
```

This type bundles all version information defined in a device.

4.6. Firmware Version Information

Each productive firmware version shall include a valid set of version information. The complete set consists of major and minor version number, version extension, vendor ID and product ID. For more detailed information about these elements refer to section 3.7.

Version information is included in the firmware by global definition of the variables listed below. If these variables are not defined by the application code, they will contain default values.

`uint32_t  ZWIR_vendorID = 0xe966`

This variable defines the Vendor ID. A vendor ID is assigned by ZMDI. It *MUST* be defined appropriately in production code!

`uint16_t  ZWIR_productID = 0`

This variable identifies a product or firmware type, respectively. It *SHALL* be defined appropriately for each firmware type. It is used by the firmware over-the-air update mechanism to distinguish different firmware types.

`uint8_t  ZWIR_firmwareMajorVersion = 0`

This variable defines the major version number of the firmware.

`uint8_t  ZWIR_firmwareMinorVersion = 0`

This variable defines the minor version number of the firmware.

`uint16_t  ZWIR_firmwareVersionExtension = 0`

This defines version extension information of the firmware.

4.7. Properties and Parameters

The behavior of built-in network stack functionality is configurable to some extend by a set of parameters which are changed using the function **ZWIR_SetParameter**.

```
int32_t
  ZWIR_SetParameter ( ZWIR_SystemParameter_t  parameter,
                      int64_t                 value )
```

This function changes the setting of a single network stack parameter. Configuration changes are getting into effect immediately when this function is called from **ZWIR_AppInitHardware**. Otherwise, the new value is buffered until **ZWIR_ResetNetwork** is called.

```
typedef enum { ... } ZWIR_SystemParameter_t
```

This enumeration names the different parameters which may be configured using **ZWIR_SetParameter**. Possible names are listed in Table 4.1, below:

Table 4.1 Configurable Stack Parameters and Their Default Values

Enumerator	Size	Default	Description
ZWIR_spRoutingTableSize	1	8	Refer to section 3.10.3
ZWIR_spNeighborCacheSize	1	8	Refer to section 3.9.3
ZWIR_spMaxSocketCount	1	8	
ZWIR_spRouteTimeout	2	3600 (s)	Refer to section 3.10.3
ZWIR_spNeighborReachableTime	2	3600 (s)	Refer to section 3.9.3
ZWIR_spMaxHopCount	1	4	Refer to section 3.10.3
ZWIR_spRouteMaxFailCount	1	3	Refer to section 3.10.3
ZWIR_spRouteRequestMinLinkRSSI	1	-128 (dBm)	Refer to section 3.10.3
ZWIR_spRouteRequestMinLinkRSSIReduction	1	0 (dB)	Refer to section 3.10.3
ZWIR_spDoDuplicateAddressDetection	1	1	Refer to section 3.8.4
ZWIR_spDoRouterSolicitation	1	1	Refer to section 3.8.3
ZWIR_spRouteRequestAttempts	1	4	Refer to section 3.10.3
ZWIR_spHeaderCompressionContext1	8	0	Refer to section 3.9.4
ZWIR_spHeaderCompressionContext2	8	0	Refer to section 3.9.4
ZWIR_spHeaderCompressionContext3	8	0	Refer to section 3.9.4

4.8. Error Codes

The error codes listed in Table 4.2 are generated by the core library and passed to the **`ZWIR_Error`** hook if it is implemented in the application code.

Table 4.2 Error Codes Generated by the Core Library

C – Identifier	Code	Default Handling
`ZWIR_eDADFailed`	100_{HEX}	Node shutdown (permanent deep-sleep, node can only be restarted with external reset)
`ZWIR_eProgExit`	101_{HEX}	Node shutdown (permanent deep-sleep, node can only be restarted with external reset)
`ZWIR_eReadMACFailed`	102_{HEX}	System reset triggered
`ZWIR_eMemoryExhaustion`[1]	103_{HEX}	System reset triggered
`ZWIR_eHostUnreachable`	104_{HEX}	Ignore – the packet causing this failure is dropped
`ZWIR_eExtClockPowerDown`	105_{HEX}	Ignore – the node will not enter power-down mode

1. *This error is only triggered when allocation fails for the memories required by the network stack. Failing allocation attempts from the application code have to be caught by checking the allocation result for NULL!*

5 UART Library Reference

The *libZWIR451x-UART1.a* and *libZWIR451x-UART2.a* libraries provide functions for easy access of the UART interfaces provided by the microcontroller. Each ZWIR451x module has two UART interfaces. ZMDI provides two separate libraries, one for each UART interface. Both libraries expose exactly the same interface. Symbol names only differ in the number behind "UART," which is part of each symbol name. Just replace the question mark in the following function and type names with 1 or 2, depending on which UART is being used.

It is possible to use only one of the UARTs or both in parallel. Consider the different priorities of the interfaces when selecting the UART (refer to section 3.6.2). The initialization of a UART interface is performed automatically if the UART library is linked into the project. The UARTs can be used after hardware initialization but not inside of **`ZWIR_AppInitHardware`**.

The UART's receive buffer is 256 bytes. If data are received on the UART interface, data are kept in the buffer until read by **`ZWIR_UART?_ReadByte`**. If the buffer is full and more data is received, **`ZWIR_Error`** is called with ***`ZWIR_eUART?Ovfl`*** as the argument. The UART libraries use an event-based programming approach. Instead of relying on polling the UART interfaces, a callback function has to be specified, which is called automatically when data is available in the receive buffer. This is done using the function **`ZWIR_UART?_SetRXCallback`**. Without calling this function, the UART receiver remains disabled, saving some power.

5.1. Symbol Reference

```
bool
  ZWIR_UART?_SendByte ( uint8_t  data )
```

Single bytes are sent via the interface using this function. The byte to be sent is provided in ***`data`***. The function returns *true* if the byte was successfully placed in the transmit buffer or *false* otherwise. If bytes are in the buffer, they are written immediately until the buffer is empty. However, be aware that significant time can elapse between a call to **`ZWIR_UART?_SendByte`** and the actual sending of the byte if there is already data in the buffer.

```
uint8_t
  ZWIR_UART?_Send ( uint8_t*  data,
                    uint16_t  dataSize )
```

A block of bytes is written to the buffer and transmission is started. The ***`data`*** argument must point to the data to be written. ***`dataSize`*** determines the number of bytes to be transferred. The return value contains the number of bytes that have actually been written. It can be lower than ***`dataSize`***.

```
bool
  ZWIR_UART?_ReadByte ( uint8_t*  data )
```

This function reads a single byte from the receive buffer. The read byte is stored to the location ***data*** points to. The function returns *true* if a byte is successfully read and *false* otherwise.

```
void
  ZWIR_UART?_SetRXCallback ( ZWIR_UART_RXCallback_t  callback )
```

This function registers a callback function that is called if data is received on the UART. The ***callback*** argument is a pointer to the function to be called. The UART receiver is not started before **ZWIR_UART?_SetRXCallback** is called. If NULL is passed as ***callback*** argument, the receiver is disabled.

```
void
  ZWIR_UART?_Setup ( uint32_t  baudRate,
                     uint32_t  parameters )
```

This function configures the UART peripheral of the microcontroller. The ***baudRate*** argument configures the speed of the transmission line. Its value must be given in bits per second. The ***parameters*** argument is a set of flags which configures the parity bit generation, the stop bit generation and controls whether flow control is used or not. The parameters argument is generated from a binary OR combination of one constants from each block described below. Default values may be omitted. In order to set all values to their default settings a zero may be passed in the ***parameters*** argument.

The following configuration options are available for parity configuration:

ZWIR_UART_NoParity	No parity bit is transmitted (default)
ZWIR_UART_OddParity	Odd parity bit is transmitted/checked
ZWIR_UART_EvenParity	Even parity bit is transmitted/checked

The following configuration options are available for stop-bit configuration:

ZWIR_UART_Stop_1	One stop-bit is transmitted at the end of a frame (default)
ZWIR_UART_Stop_2	Two stop bits are transmitted at the end of a frame

The following configuration options are available for flow-control configuration:

ZWIR_UART_NoFlowControl	Flow control is disabled (default)
ZWIR_UART_HWFlowControl	Use hardware flow control with CTS and RTS

Note: A call to this function drops all bytes which are still in the transmission buffer. If this function is called during an active transmission, the active transmission will fail very likely.

Note: When flow control is enabled the configuration of the CTS and RTS pins of the corresponding UART interface are configured as input and alternative push/pull output, respectively. This configuration overwrites the configuration these pins had before. When flow control changes from enabled to disabled, the pin configuration of the CTS and RTS pins is not changed.

```
bool
  ZWIR_UART?_IsTXEmpty ( void )
```

This function returns false if there are still bytes in the UART transmit buffer and returns true otherwise.

```
uint16_t
  ZWIR_UART?_GetAvailableTXBuffer ( void )
```

This function returns the number of free bytes in the UART TX buffer.

```
typedef
  void ( * ZWIR_UART_RXCallback_t ) ( void )
```

This is the type definition for a UART callback function. This type is used with **ZWIR_UART?_SetRXCallback**. The callback does not accept any elements and does not return.

ZWIR_UART?_PRINTF

This is a macro provided for convenience. If high-level functions for text output like **printf** are used, an appropriate low-level function must be provided, which can output characters to a device. This macro defines a low-level output function writing to the UART interface. This macro must only be used once in the whole application source code. It must not be put inside of function definitions and should not be put in header files. It is not possible to use both, **ZWIR_UART1_PRINTF** and **ZWIR_UART2_PRINTF** in the same project.

5.2. Error Codes

The error codes listed in Table 5.1 are generated by the UART libraries and passed to the **ZWIR_Error** hook if it is implemented in the application code.

Table 5.1 Error Codes Generated by the UART Libraries

C – Identifier	Code	Default Handling
libZWIR451x-UART1.a		
ZWIR_UART1_eOvfl	200_{HEX}	Ignore
ZWIR_UART1_eParity	201_{HEX}	Ignore
ZWIR_UART1_eFrame	202_{HEX}	Ignore
ZWIR_UART1_eNoise	203_{HEX}	Ignore
libZWIR451x-UART2.a		
ZWIR_UART2_eOvfl	210_{HEX}	Ignore
ZWIR_UART2_eParity	211_{HEX}	Ignore
ZWIR_UART2_eFrame	212_{HEX}	Ignore
ZWIR_UART2_eNoise	213_{HEX}	Ignore

6 GPIO Library Reference

The GPIO library provides a convenient interface for accessing and controlling the GPIO ports of the module. It allows configuring GPIOs to be used as application programmable inputs or outputs and it is possible to enable or disable special function of certain ports such as the JTAG ports. All functions from the GPIO library are also accessible using the MCUs peripheral control registers.

The intention of the GPIO library is to provide a high-level, lightweight, convenient interface to the GPIOs. For that reason the GPIO library functions do no implement parameter checking. It is in the responsibility of the application to ensure that appropriate parameters are used. All microcontroller GPIO pins which are not connected to one of the modules I/O pins are locked, so that it is impossible to change their configuration accidentally.

6.1. Symbol Reference

```
void
  ZWIR_GPIO_ConfigureAsOutput ( ZWIR_GPIO_Pin_t             pins,
                                ZWIR_GPIO_DriverStrength_t  driver,
                                ZWIR_GPIO_OutputMode_t      mode )
```

This function registers one or multiple pins as output. In the ***pins*** argument specifies the pin to be configured. If multiple pins need to be configured provide a binary or'ed combination of the enumeration values corresponding to the pins. The ***driver*** argument determines the driving strength of the pin; the ***mode*** argument determines the output mode. If multiple pins are specified, all pins will be configured in the same way.

Be sure to use a combination of **ZWIR_GPIO_Pin_t** enumeration values to specify which pins shall be configured. Otherwise the wrong pins might be configured resulting in a configuration which could cause damage to the system.

```
void
  ZWIR_GPIO_ConfigureAsInput ( ZWIR_GPIO_Pin_t        pins,
                               ZWIR_GPIO_InputMode_t  mode )
```

This function registers one or multiple pins as input. The ***pins*** argument specifies the pin to be configured. If multiple pins need to be configured, provide a binary or'ed combination of the enumeration values corresponding to the pins. The ***mode*** argument selects the configuration of the inputs. If multiple pins are specified, all pins will be configured in the same way.

Be sure to use a combination of **ZWIR_GPIO_Pin_t** enumeration values to specify which pins shall be configured. Otherwise the system might behave not as expected.

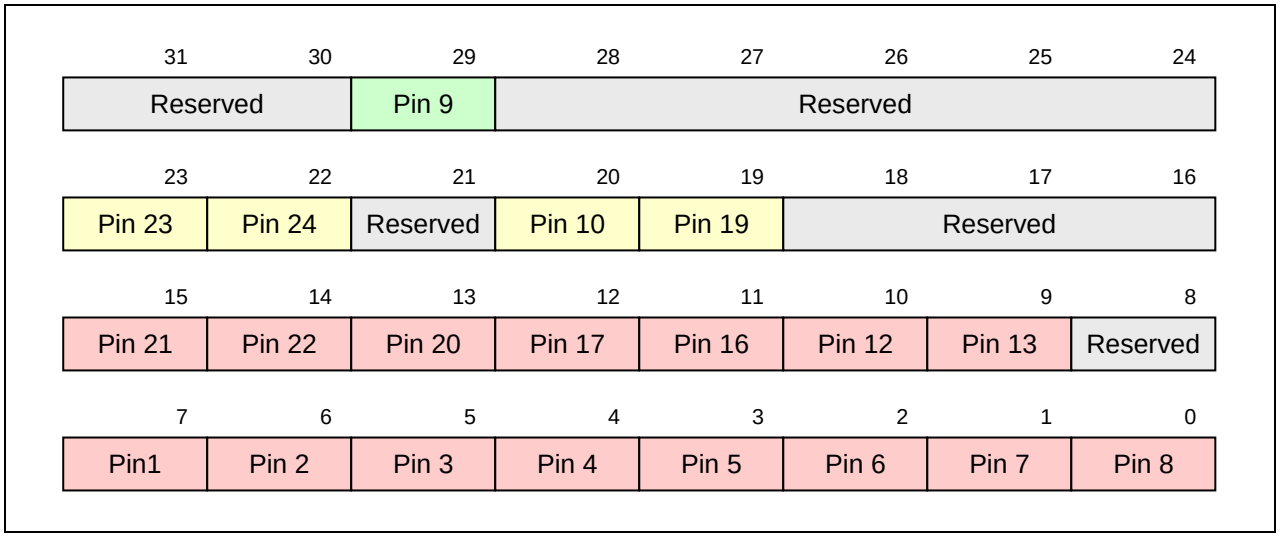
```
bool
  ZWIR_GPIO_Read ( ZWIR_GPIO_Pin_t  pin )
```

This function reads the input value of a single pin. The function doesn't care if the pin is configured as input or output pin. If the pin is configured as analog input, the return value is undefined.

```
uint32_t
  ZWIR_GPIO_ReadMultiple ( ZWIR_GPIO_Pin_t  pins )
```

This function reads the input value of multiple pins. The function doesn't care if the ***pins*** are configured as inputs or outputs. If a pin is configured as analog input, the return value at the corresponding bit is undefined. The result is aligned as shown in Figure 6.1. In order to extract single bit results, the return value of the function may be binary or'ed with the **ZWIR_GPIO_Pin_t** enumeration values.

Figure 6.1 ZWIR_GPIO_ReadMultiple Result Alignment



Note: Only pins from the same GPIO bank are read at the same time. If pins don't share the same GPIO bank, there will be a time difference between the accesses to their input registers. All pins belonging to the same GPIO bank are highlighted with the same color in Figure 6.1.

```
void
  ZWIR_GPIO_Write ( ZWIR_GPIO_Pin_t  pins,
                    bool             value )
```

This function sets the output value of one or multiple pins. All pins specified in the ***pins*** argument are set to ***value***. The output is written regardless of the pin configuration! If one of the pins was configured as pull-up or pull-down input, writing the output register of this pin can change the pull-up/pull-down configuration accidentally!

```
void
  ZWIR_GPIO_Remap ( ZWIR_GPIO_RemapFunction_t  function,
                    int32_t                    value )
```

This function is used to control the remapping of GPIO pins to system functions. It allows configuring whether the JTAG pins are used for debug purposes or as normal GPIO pins. The ***value*** argument shall be one of the options defined by the enumeration type **ZWIR_GPIO_SWJRemapValue_t**.

typedef enum { ... } ZWIR_GPIO_Pin_t

This enumeration type assigns a name to each pin. Some GPIO operations allow specifying multiple pins. This is done by or-combining the pins.

ZWIR_Pin1	MCU port A7
ZWIR_Pin2	MCU port A6
ZWIR_Pin3	MCU port A5
ZWIR_Pin4	MCU port A4
ZWIR_Pin5	MCU port A3
ZWIR_Pin6	MCU port A2
ZWIR_Pin7	MCU port A1
ZWIR_Pin8	MCU port A0
ZWIR_Pin9	MCU port C13
ZWIR_Pin12	MCU port A10
ZWIR_Pin13	MCU port A9
ZWIR_Pin16	MCU port A11
ZWIR_Pin17	MCU port A12
ZWIR_Pin19)	MCU port B3 (remapped function; default configuration JTDO
ZWIR_Pin20)	MCU port A13 (remapped function; default configuration JTMS
ZWIR_Pin21	MCU port A15 (remapped function; default configuration: JTDI)
ZWIR_Pin22	MCU port A14 (remapped function; default configuration: JTCK)
ZWIR_Pin23	MCU port B7
ZWIR_Pin24	MCU port B6

typedef enum { ... } ZWIR_GPIO_DriverStrength_t

This enumeration value specifies the driving strength of GPIO output pins.

ZWIR_GPIO_dsLow	Low driving strength
ZWIR_GPIO_dsMedium	Medium driving strength
ZWIR_GPIO_dsHigh	High driving strength

typedef enum { ... } ZWIR_GPIO_OutputMode_t

This enumeration value specifies the mode of GPIO output pins.

ZWIR_GPIO_omPushPull	Application controlled push/pull output
ZWIR_GPIO_omOpenDrain	Application controlled open drain output
ZWIR_GPIO_omAlternativePushPull	Hardware controlled push/pull output
ZWIR_GPIO_omAlternativeOpenDrain	Hardware controlled open drain output

typedef enum { ... } ZWIR_GPIO_InputMode_t

This enumeration value specifies the mode of GPIO input pins.

ZWIR_GPIO_imAnalog	Analog input (default configuration)
ZWIR_GPIO_imFloating	Floating input
ZWIR_GPIO_imPullUp	Pull-up input
ZWIR_GPIO_imPullDown	Pull-down input

typedef enum { ... } ZWIR_GPIO_RemapFunction_t

This enumeration type is used to specify which remapping shall be changed with **ZWIR_GPIO_Remap**.

ZWIR_GPIO_rfSWJ	Configure remapping of the JTAG/SWD pins (Pin19 – Pin 22)

typedef enum { ... } ZWIR_GPIO_SWJRemapValue_t

In calls to **ZWIR_GPIO_Remap** this enumeration value specifies which configuration is used for JTAG/SWD remapping.

ZWIR_GPIO_swjrEnableSWJ	Enable full JTAG/SWD support (pins 19 to 22 cannot be used as GPIO).
ZWIR_GPIO_swjrSWOnly	Enable SWD support only (pins 20 and 22 cannot be used as GPIO).
ZWIR_GPIO_swjrDisableSWJ	Disable JTAG/SWD support (pins 19 to 22 can be used as GPIO).

7 IPSec Library Reference

The IPSec library provides functions to manage security policies and security associations. A security policy is added using **ZWIRSEC_AddSecurityPolicy**. Security Associations can be added using the function **ZWIRSEC_AddSecurityAssociation**. For more detailed information about security policies and security association, refer to the ZMDI application note “Using IPSec and IKEv2 in 6LoWPANs.”

7.1. Symbol Reference

```
uint8_t
  ZWIRSEC_AddSecurityPolicy ( ZWIRSEC_PolicyType_t            type,
                              ZWIR_IPv6Address_t*             remoteAddress,
                              uint8_t                         prefix,
                              ZWIR_Protocol_t                 proto,
                              uint16_t                        lowerPort,
                              uint16_t                        upperPort,
                              ZWIRSEC_SecurityAssociation_t*  securityAssociation )
```

A call to this function adds a security policy to the IPSec security policy database. The ***type*** argument determines the traffic direction and how packets have to be handled. The combination of the ***remoteAddress***, ***prefix***, ***protocol***, ***lowerPort*** and ***upperPort*** arguments specify the traffic which is affected by this policy. See section 3.12 and ZMDI application note “Using IPSec and IKEv2 in 6LoWPANs” for more details. The function returns the security policy index of the newly created security policy. In case of error FF_{HEX} is returned.

The last argument specifies the security association to be used by this policy. A security association must be created using **ZWIRSEC_AddSecurityAssociation** before **ZWIRSEC_AddSecurityPolicy** is called. If IKEv2 should be used for generating the security association automatically, pass NULL as the ***securityAssociation*** argument.

```
void
  ZWIRSEC_RemoveSecurityPolicy ( uint8_t  spi )
```

This function removes the security policy with index ***spi*** from the security policy database. If no index is stored at ***spi*** the function does nothing.

```
ZWIRSEC_SecurityAssociation_t*
  ZWIRSEC_AddSecurityAssociation ( uint32_t                      securityParamIdx,
                                   ZWIRSEC_EncryptionSuite_t*    encSuite,
                                   ZWIRSEC_AuthenticationSuite_t* authSuite )
```

This function adds a security association to the security association database manually. Use this function before calling **ZWIRSEC_AddSecurityPolicy** if not using IKEv2 for automatic key exchange.

The ***securityParamIdx*** argument is a unique number identifying the security association. The ***encSuite*** and ***authSuite*** parameters specify the encryption and authentication algorithms and keys. Refer to section 3.12.1 and the links for **ZWIRSEC_EncryptionSuite_t** and **ZWIRSEC_AuthenticationSuite_t** for more details.

The function returns a pointer to the security association descriptor if it was created successfully. In case of error the function returns NULL.

```
void
  ZWIRSEC_RemoveSecurityAssociation ( ZWIRSEC_SecurityAssociation_t*  sa )
```

This function removes the security association pointed to by ***sa***.

```
typedef enum { ... }  ZWIRSEC_PolicyType_t
```

IPSec policy type enumeration. Possible values include

ZWIRSEC_ptOutputApply	**0x11**	Secure outbound traffic with IPSec
ZWIRSEC_ptOutputBypass	**0x12**	Bypass outbound traffic unsecured
ZWIRSEC_ptOutputDrop	**0x13**	Drop outbound traffic
ZWIRSEC_ptInputApply	**0x21**	Unsecure inbound traffic with IPSec
ZWIRSEC_ptInputBypass	**0x22**	Bypass inbound traffic unsecured
ZWIRSEC_ptInputDrop	**0x23**	Drop inbound traffic

```
typedef enum { ... }  ZWIR_Protocol_t
```

IPSec protocol enumeration. Possible values include

ZWIR_protoAny	**0**
ZWIR_protoTCP	**6**
ZWIR_protoUDP	**17**
ZWIR_protoICMPv6	**58**

```
typedef struct {
    ZWIRSEC_EncryptionoAlgorithm_t  algorithm
    uint8_t                         key   [ 16 ]
    uint8_t                         nonce [  4 ]
} ZWIRSEC_EncryptionSuite_t
```

This structure carries all encryption related information. It is used to pass encryption information to **ZWIRSEC_AddSecurityAssociation**.

```
typedef struct {
    ZWIRSEC_AuthenticationAlgorithm_t  algorithm
    uint8_t                            key [ 16 ]
} ZWIRSEC_AuthenticationSuite_t
```

This structure carries all authentication related information. It is used to pass authentication information to **ZWIRSEC_AddSecurityAssociation**.

typedef void* ZWIRSEC_SecurityAssociation_t

Objects of this type are returned by **ZWIRSEC_AddSecurityAssociation**. They are passed to **ZWIRSEC_AddSecurityPolicy**.

typedef enum { ... } ZWIRSEC_EncryptionAlgorithm_t

Enumeration of algorithms available for encryption; possible values include

ZWIRSEC_encNull	**11**	no encryption
ZWIRSEC_encAESCTR	**13**	AES[1] Counter Mode based encryption

typedef enum { ... } ZWIRSEC_AuthenticationAlgorithm_t

Enumeration of algorithms available for authentication; possible values include

ZWIRSEC_authNull	**0**	no authentication
ZWIRSEC_authAESXCBC96	**5**	Extended AES128 CBC[2] Mode based auth.

[1] AES – Advanced Encryption Standard
[2] CBC – Cyclic Block Cipher

8 IKEv2 Library Reference

IKEv2 is used for IPSec key management. Using IKEv2 it is possible to limit the lifetime of a security association and automatically regenerate it with new keys automatically. In order to add IKEv2 functionality to an application, the IKEv2 library must be linked into the project. If this is done, the IKEv2 daemon is automatically registered as the key management engine for IPSec.

The only task to be done by the application is adding the suitable authentication entries to the IKEv2 authentication database. This is done using the **`ZWIRSEC_AddIKEAuthenticationEntry`** function.

8.1. Symbol Reference

```
uint8_t
  ZWIRSEC_AddIKEAuthenticationEntry (  ZWIR_IPv6Address_t*  remoteAddress,
                                       uint8_t              prefixLength,
                                       uint8_t*             id,
                                       uint8_t              idLength,
                                       uint8_t*             presharedKey )
```

Calling this function adds an authentication entry to the IKE authentication database. The ***`remoteAddress`*** argument contains the IPv6 address of the remote device; ***`prefixLength`*** contains the prefix length of ***`remoteAddress`***. The device identifier is given in ***`id`***. ***`idLength`*** specifies its length. The ***`presharedKey`*** argument carries a pointer to the pre shared key that is used for authentication.

The function returns *true* on success or *false* otherwise. A *false* return indicates there is no room in the authentication database.

```
uint8_t  ZWIRSEC_ikeRetransmitTime = 10
```

Weak constant defining how many seconds IKE waits for a reply before retransmission is initiated. The time should be long enough to enable IKE processing at the receiver. This value largely depends on the clock frequency. Set the value accordingly. The predefined value of 10 seconds is suitable for a receiver clock frequency of 32 MHz or 64 MHz only. The value can be redefined by definition of the variable **`ZWIR_ikeRetransmitTime`** with an appropriate value in the application code.

```
uint32_t  ZWIRSEC_ikeRekeyTime = 86400
```

This is a weakly defined variable that controls the interval at which the IKE connection must be rekeyed. The default setting corresponds to one week. In order to change this value, the variable **`ZWIRSEC_ikeRekeyTime`** must be defined with an appropriate value in the application code.

```
uint32_t  ZWIRSEC_ikeSARekeyTime = 604800
```

This weakly defined variable controls the interval during which security associations remain valid before rekeying is required. The default setting corresponds to one day. In order to change this value, the variable **`ZWIRSEC_ikeSARekeyTime`** must be defined with an appropriate value in the application code.

8.2. Library Parameters

Table 8.1 shows a summary of ZMDI's IKEv2 library parameters and properties.

Table 8.1 Overview of IKEv2 Library Parameters and Properties

Property	Value
Authentication database size	5
Number of used sockets	2
Parameter	**Value**
ZWIRSEC_ikeSARekeyTime	604800
ZWIRSEC_ikeRekeyTime	86400
ZWIRSEC_ikeRetransmitTime	10 s

9 Over-the-Air Update Library

This library implements the Firmware Over-the-Air Update functionality. The only function exported from this library is used to register the Over-the-Air Update daemon in the system.

9.1. Library Reference

```
void
  ZWIR_OTAU_Register ( uint16_t  port )
```

Register the Over-the-Air Update daemon and configures the UDP port the daemon is listening on.

```
typedef enum { ... } ZWIR_OTAU_ErrorCode_t
```

Defines error codes which have to be handled locally. Theses error codes are used with **ZWIR_Error**. Possible values are:

ZWIR_eInvalidVID this error is reported when the configured vendor ID is invalid

10 Accessing Microcontroller Resources

Many applications might wish to make use of the rich internal resources provided by the microcontroller. In general this is no problem, but some caution must be taken when this is considered. No resources must be used that are already occupied by the operating system. Furthermore some of the MCU configuration parameters must not be altered. Refer to the next section for a complete list of resources that are used by the OS.

The library does not provide dedicated functions for configuration of the microcontroller peripherals. This must be done by third-party libraries, or by programming the appropriate configuration registers directly. Names for interrupt handlers are predefined by the library. If interrupt handlers are required, a function with the library-determined name must be implemented by the library user.

10.1. Internal Microcontroller Configuration

The STM32 is clocked from its internal 8 MHz oscillator (HSI). The system clock (SYSCLK) is taken from the phase-locked loop (PLL) output (PLLCLK). The PLL source is HSI/2 and the PLL multiplier is 16, so SYSCLK has a frequency of 64 MHz. The Advanced High-performance Bus (AHB) clock (HCLK) is configured according to the selected CPU frequency (see **`ZWIR_SetFrequency`**). APB1 clock (PCLK1) frequency is always 4 MHz, APB2 clock (PCLK2) frequency is always 8 MHz. It is strongly recommended that the frequencies of APB1 or APB2 are *not* changed! Doing so would result in wrong timing behavior of the operating system and might even result in system breakdown.

The Cortex System Timer (SysTick) is used as the operating system base timer. It is configured to issue an interrupt each millisecond. The real-time clock (RTC) is used for the different sleep modes.

All microcontroller GPIO pins which are not connected to one of the modules I/O pins are locked, so that it is impossible to change their configuration accidentally. The configuration of the external interrupt line 0 (EXTI0) and the configuration of the SPI2 peripheral of the microcontroller must not be changed. Otherwise the interfacing between MCU and transceiver might be impaired.

10.2. Interrupt Handlers

The API library comes with a set of predefined interrupt handlers that is sufficient for the built-in functionality but does not go beyond it. For all other interrupts that are not required, default handlers are provided that typically do nothing or perform a reset in case of an error. Most of the interrupts are defined as weak symbols in the library. This means that the default implementations of the handlers can be overwritten by simply defining the interrupt handler symbol in the user's application code. Only those interrupts that are required for proper operation of the stack are not defined as weak and therefore cannot be overwritten. An attempt to overwrite these handlers will result in a linker error.

Table 10.1 lists the interrupts for the STM32. For each interrupt, the handler name, the default priority and the default behavior is shown and whether or not the interrupt can be overwritten.

Table 10.1 STM32 Interrupt Vector Table

Interrupt		Implementation			
Id	**Name**	**Handler**	**Prio**	**Fix**	**Default-Behavior**
0	Reset	Reset_Handler	-3	Yes	Perform system reset
1	NMI	ZWIR_ISR_NMI	-2	No	Perform system reset
2	HardFault	ZWIR_ISR_HardFault	-1	No	Perform system reset
3	MemManage	ZWIR_ISR_MemManage	0	No	Perform system reset
4	BusFault	ZWIR_ISR_BusFault	1	No	Perform system reset
5	UsageFault	ZWIR_ISR_UsageFault	2	No	Perform system reset
6	SVCall	ZWIR_ISR_SVCall	3	No	NULL
7	DebugMonitor	ZWIR_ISR_DebugMonitor	4	No	NULL
8	PendSV	ZWIR_ISR_PendSV	5	No	NULL
9	SysTick	ZWIR_ISR_SysTick	6	Yes	Used as operating system timer
10	WWDG	ZWIR_ISR_WWDG	7	No	NULL
11	PVD	ZWIR_ISR_PVD	8	No	Perform system reset
12	TAMPER	ZWIR_ISR_TAMPER	9	No	NULL
13	RTC	ZWIR_ISR_RTC	10	Yes	Reserved for OS use
14	FLASH	ZWIR_ISR_FLASH	11	No	NULL
15	RCC	ZWIR_ISR_RCC	12	No	NULL
16	EXTI0	ZWIR_ISR_EXTI0	13	Yes	Handle transceiver service request
17	EXTI1	ZWIR_ISR_EXTI1	14	No	NULL
18	EXTI2	ZWIR_ISR_EXTI2	15	No	NULL
19	EXTI3	ZWIR_ISR_EXTI3	16	No	NULL
20	EXTI4	ZWIR_ISR_EXTI4	17	No	NULL
21	DMA1_Channel1	ZWIR_ISR_DMA1_Channel1	18	No	NULL
22	DMA1_Channel2	ZWIR_ISR_DMA1_Channel2	19	No	NULL
23	DMA1_Channel3	ZWIR_ISR_DMA1_Channel3	20	No	NULL
24	DMA1_Channel4	ZWIR_ISR_DMA1_Channel4	21	No	NULL
25	DMA1_Channel5	ZWIR_ISR_DMA1_Channel5	22	No	NULL
26	DMA1_Channel6	ZWIR_ISR_DMA1_Channel6	23	No	NULL
27	DMA1_Channel7	ZWIR_ISR_DMA1_Channel7	24	No	NULL
28	ADC1_2	ZWIR_ISR_ADC1_2	25	No	NULL
29	USB_HP_CAN_TX	ZWIR_ISR_USB_HP_CAN1_TX	26	No	NULL
30	USB_LP_CAN_RX0	ZWIR_ISR_USB_LP_CAN1_RX0	27	No	NULL
31	CAN_RX1	ZWIR_ISR_CAN1_RX1	28	No	NULL
32	CAN_SCE	ZWIR_ISR_CAN1_SCE	29	No	NULL

Interrupt		Implementation			
Id	**Name**	**Handler**	**Prio**	**Fix**	**Default-Behavior**
33	EXTI9_5	ZWIR_ISR_EXTI9_5	30	No	NULL
34	TIM1_BRK	ZWIR_ISR_TIM1_BRK	31	No	NULL
35	TIM1_UP	ZWIR_ISR_TIM1_UP	32	No	NULL
36	TIM1_TRG_COM	ZWIR_ISR_TIM1_TRG_COM	33	No	NULL
37	TIM1_CC	ZWIR_ISR_TIM1_CC	34	No	NULL
38	TIM2	ZWIR_ISR_TIM2	35	No	NULL
39	TIM3	ZWIR_ISR_TIM3	36	No	NULL
40	TIM4	ZWIR_ISR_TIM4	37	Yes	NULL
41	I2C1_EV	ZWIR_ISR_I2C1_EV	38	No	NULL
42	I2C1_ER	ZWIR_ISR_I2C1_ER	39	No	NULL
43	I2C2_EV	ZWIR_ISR_I2C2_EV	40	No	NULL
44	I2C2_ER	ZWIR_ISR_I2C2_ER	41	No	NULL
45	SPI1	ZWIR_ISR_SPI1	42	No	NULL
46	SPI2	ZWIR_ISR_SPI2	43	Yes	Used by network stack
47	USART1	ZWIR_ISR_USART1	44	No	NULL[3]
48	USART2	ZWIR_ISR_USART2	45	No	NULL[4]
49	USART3	ZWIR_ISR_USART3	46	No	NULL
50	EXTI15_10	ZWIR_ISR_EXTI15_10	47	No	NULL
51	RTCAlarm	ZWIR_ISR_RTCAlarm	48	Yes	Reserved for OS use
52	USBWakeup	ZWIR_ISR_USBWakeup	49	No	NULL
53	TIM8_BRK	ZWIR_ISR_TIM8_BRK	50	No	NULL
54	TIM8_UP	ZWIR_ISR_TIM8_UP	51	No	NULL
55	TIM8_TRG_COM	ZWIR_ISR_TIM8_TRG_COM	52	No	NULL
56	TIM8_CC	ZWIR_ISR_TIM8_CC	53	No	NULL
57	ADC3	ZWIR_ISR_ADC3	54	No	NULL
58	FSMC	ZWIR_ISR_FSMC	55	No	NULL
59	SDIO	ZWIR_ISR_SDIO	56	No	NULL
60	TIM5	ZWIR_ISR_TIM5	57	No	NULL
61	SPI3	ZWIR_ISR_SPI3	58	No	NULL

[3] Implementation is provided if libZWIR451x-UART0.a is linked into the project
[4] Implementation is provided if libZWIR451x-UART1.a is linked into the project

Interrupt		Implementation			
Id	**Name**	**Handler**	**Prio**	**Fix**	**Default-Behavior**
62	UART4	ZWIR_ISR_UART4	59	No	NULL
63	UART5	ZWIR_ISR_UART5	60	No	NULL
64	TIM6	ZWIR_ISR_TIM6	61	No	NULL
65	TIM7	ZWIR_ISR_TIM7	62	No	NULL
66	DMA2_Channel1	ZWIR_ISR_DMA2_Channel1	63	No	NULL
67	DMA2_Channel2	ZWIR_ISR_DMA2_Channel2	64	No	NULL
68	DMA2_Channel3	ZWIR_ISR_DMA2_Channel3	65	No	NULL
69	DMA2_Channel4_5	ZWIR_ISR_DMA2_Channel4_5	66	No	NULL

10.3. Default I/O Configuration

Table 10.2 shows the default I/O configuration that is set when the device is powered on and no manual changes are made to the I/Os. The left section shows the configuration if only *libZWIR45xx-6LoWPAN.a* is linked into the program; the right section shows changes applied when additional libraries are linked.

Table 10.2 STM32 Default I/O Configuration

MCU	Module	libZWIR45xx-6LoWPAN.a		Alternative Configuration		
Port	**Pin**	**Configuration**	**Drive**	**Configuration**	**Drive**	**Library**
A0	8	Analog Input	-	None		
A1	7	Analog Input	-	None		
A2	6	Analog Input	-	2 MHz Push/Pull Alternative Output	x	libZWIR451x-UART2.a
A3	5	Analog Input	-	Floating Alternative Input	-	libZWIR451x-UART2.a
A4	4	Analog Input	-	None		
A5	3	Analog Input	-	None		
A6	2	Analog Input	-	None		
A7	1	Analog Input	-	None		
A8	-	2 MHz Push/Pull Output	0	None		
A9	13	Analog Input	-	2 MHz Push/Pull Alternative Output	x	libZWIR451x-UART1.a
A10	12	Analog Input	-	Floating Alternative Input	-	libZWIR451x-UART1.a
A11	16	Analog Input	-	None		
A12	17	Analog Input	-	None		
A13	-	Floating Input	-	None		
A14	-	Floating Input	-	None		
A15	-	Floating Input	-	None		

MCU	Module	libZWIR45xx-6LoWPAN.a		Alternative Configuration		
Port	Pin	Configuration	Drive	Configuration	Drive	Library
B0	-	Pull-up Input	-	None		
B1	-	Floating Input	-	None		
B2	-	Floating Input	-	None		
B3	-	Floating Input	-	None		
B4	-	Floating Input	-	None		
B5	-	Floating Input	-	None		
B6	24	Floating Input	-	None		
B7	23	Floating Input	-	None		
B8	-	Floating Input	-	None		
B9	-	2 MHz Push/Pull Output	1	None		
B10	-	2 MHz Push/Pull Output	0	None		
B11	-	2 MHz Push/Pull Output	0	None		
B12	-	10 MHz Push/Pull Output	1	None		
B13	-	10 MHz Push/Pull Alternative Output	x	None		
B14	-	10 MHz Push/Pull Alternative Output	x	None		
B15	-	10 MHz Push/Pull Alternative Output	x	None		
C0	-	2 MHz Push/Pull Output	1	None		
C1	-	2 MHz Push/Pull Output	1	None		
C2	-	2 MHz Push/Pull Output	1	None		
C4	-	Analog Input	-	None		
C5	-	Analog Input	-	None		
C6	-	Analog Input	-	None		
C7	-	Analog Input	-	None		
C8	-	Analog Input	-	None		
C9	-	Analog Input	-	None		
C10	-	Analog Input	-	None		
C11	-	Analog Input	-	None		
C12	-	Analog Input	-	None		
C13	9	Analog Input	-	None		
C14	-	Analog Input	-	None		
C15	-	Analog Input	-	None		

11 Certification

11.1. European R&TTE Directive Statements

The ZWIR4512 module has been tested and found to comply with Annex IV of the R&TTE Directive 1999/5/EC and is subject of a notified body opinion. The module has been approved for Antennas with gains of 4 dBi or less.

11.2. Federal Communication Commission Certification Statements

11.2.1. Statements

This equipment has been tested and found to comply with the limits for a **Class B digital device**, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference in a residential installation. This equipment generates, uses and can radiate radio frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation. If this equipment does cause harmful interference to radio or television reception, which can be determined by turning the equipment off and on, the user is encouraged to try to correct the interference by one or more of the following measures:

- Reorient or relocate the receiving antenna.
- Increase the separation between the equipment and receiver.
- Connect the equipment into an outlet on a circuit different from that to which the receiver is connected.
- Consult the dealer or an experienced radio/TV technician for help.

This device complies with part 15 of the FCC Rules. Operation is subject to the following two conditions: (1) This device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

Modifications not expressly approved by ZMD AG could void the user's authority to operate the equipment.

The internal / external antennas used for this mobile transmitter must provide a separation distance of at least 20 cm from all persons and must not be co-located or operating in conjunction with any other antenna or transmitter.

11.2.2. Requirements

The ZWIR4512 complies with Part 15 of the FCC rules and regulations. In order to retain compliance with the FCC certification requirements, the following conditions must be met:

1. Modules must be installed by original equipment manufacturers (OEM) only
2. The module must only be operated with antennas …i
3. The OEM must place a clearly visible text label on the outside of the end-product containing the text shown in Figure 8-1, below.

Figure 11.1 FCC Compliance Statement to be printed on Equipment Incorporating ZWIR4512 Devices

Contains FCC ID: COR-ZWIR4512AC1
This device complies with part 15 of the FCC Rules. Operation is subject to the following two conditions: (1) This device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

11.2.3. Accessing the FCC ID

ZWIR451x modules are capable of showing their FCC-ID electronically. C-API applications may read the modules FCC-ID through the function **`ZWIR_GetFCCID`**. Due to space constraints the FCC ID is not printed on the module. Host devices incorporating this module must be marked according to above guidelines.

11.3. Supported Antennas

The FCC compliance testing of the ZWIR4512 has been carried out using the MEXE902RPSM antenna from PCTEL Inc. This antenna has an omnidirectional radiation pattern at an antenna gain of 2 dBi. In order to be allowed to use the module without re-certification, the product incorporating the ZWIR4512 module must either use the antenna mentioned above or must use an antenna with an omnidirectional radiation pattern and a gain being less than or equal to 2 dBi.

12 Alphabetical List of Symbols

13 Related Documents

IETF Documents	Source
Internet Protocol, Version 6 (IPv6) Specification	*RFC 2460, http://tools.ietf.org/html/rfc2460*
IP Version 6 Addressing Architecture	*RFC 4291, http://tools.ietf.org/html/rfc4291*
Security Architecture for the Internet Protocol	*RFC 4301, http://tools.ietf.org/html/rfc4301*
Internet Key Exchange (IKEv2) Protocol	*RFC 5996, http://tools.ietf.org/html/rfc5996*
Neighbor Discovery for IP Version 6 (IPv6)	*RFC 4861, http://tools.ietf.org/html/rfc4861*
IPv6 Stateless Address Auto-configuration	*RFC 4862, http://tools.ietf.org/html/rfc4862*
Transmission of IPv6 Packets over IEEE 802.15.4 Networks	*RFC 4944, http://tools.ietf.org/html/rfc4944*
ZMDI Documents	**File Name**
ZWIR4512 Data Sheet	*ZWIR4512_Data_Sheet_revX.xy.pdf*
ZWIR451x Application Note: Using IPSec and IKEv2 in 6LoWPANs	*ZWIR45xx_AN_Security_revX.xy.pdf*
ZWIR451x Application Note: Enabling Firmware Over-the-Air Updates	*ZWIR451x_AN_OTAU_revX.xy.pdf*

Visit ZMDI's website www.zmdi.com or contact your nearest sales office for the latest version of these documents.

14 Glossary

Term	Description
6LoWPAN	IPv6 over Low Power Wireless Personal Area Networks
AES	Advanced Encryption Standard
AH	Authentication Header
API	Application Programming Interface
ARP	Address Resolution Protocol
CBC	Cyclic Block Cipher
DAD	Duplicate Address Detection
DHCP	Dynamic Host Configuration Protocol
ESP	Encapsulating Security Payload
GPIO	General Purpose Input/Output
IETF	Internet Engineering Task Force
IKEv2	Internet Key Exchange version 2
IPSec	Internet Protocol Security
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6

Term	Description
MAC	Media Access Control
LAN	Local Area Network
MCU	Micro Controller Unit
NDP	Neighbor Discovery Protocol
NA	Neighbor Advertisement
NS	Neighbor Solicitation
OSI	Open Systems Interconnection
PAN	Personal Area Network
PLL	Phase-Locked Loop
PSK	Pre Shared Key
RA	Router Advertisement
RS	Router Solicitation
RSSI	Receive Signal Strength Indicator
SA	Security Association
SAD	Security Association Database
SP	Security Policy
SPD	Security Policy Database
SWD	Serial Wire Debug
TRX	Transceiver
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
WAN	Wide Area Network
WPAN	Wireless Personal Area Network

15 Document Revision History

Revision	Date	Description
1.00	October 1, 2010	Initial Version
1.10	December 13, 2010	- Added I/O pin descriptions for SAM3S based modules (e.g. ZWIR4522-I) - Added interrupt vector table for ZWIR452x-I modules (e.g. ZWIR4511-I) - Corrected declaration of ZWIR_DiscoveryCallback_t - Replaced invalid declaration ZWIR_GetPacketRSSI with ZWIR_GetLastRSSI - Applied some beautification
1.20	March 27, 2011	- Renamed document to ZWIR451x Programming Guide - Removed parts of documentation dedicated to SAM3S based modules - Adapted documentation to library release 1.2 - Cross-linked all symbols
1.30	July 5, 2011	Minor revisions for clarity
1.40	November 17, 2011	- Added libGPIO documentation - Minor revisions for clarity
1.60	June 18, 2012	- Added documentation of new functionality provided with API version 1.6 - Many text improvements for clarity - Fixed error in Table 2.1 - Added documentation for NetMA functions and types
1.61	July 27, 2012	-Minor edits
1.62	August 31, 2012	-Added documentation of FCC-ID readout command -Added R&TTE & FCC conformity statements

Sales and Further Information www.zmdi.com wpan@zmdi.com

Zentrum Mikroelektronik Dresden AG
Grenzstrasse 28
01109 Dresden
Germany

Phone +49.351.8822.7476
Fax +49.351.8822.87476

ZMD America, Inc.
1525 McCarthy Blvd., #212
Milpitas, CA 95035-7453
USA

Phone +855-ASK-ZMDI
(+855.275.9634)

Zentrum Mikroelektronik Dresden AG, Japan Office
2nd Floor, Shinbashi Tokyu Bldg.
4-21-3, Shinbashi, Minato-ku
Tokyo, 105-0004
Japan

Phone +81.3.6895.7410
Fax +81.3.6895.7301

ZMD FAR EAST, Ltd.
3F, No. 51, Sec. 2,
Keelung Road
11052 Taipei
Taiwan

Phone +886.2.2377.8189
Fax +886.2.2377.8199

Zentrum Mikroelektronik Dresden AG, Korean Office
POSCO Centre Building
West Tower, 11th Floor
892 Daechi, 4-Dong,
Kangnam-Gu
Seoul, 135-777
Korea

Phone +82.2.559.0660
Fax +82.2.559.0700