ZigBeeNet

Developer's Guide

© 2008 MeshNetics

© 2008 MeshNetics. All rights reserved.

No part of the contents of this manual may be transmitted or reproduced in any form or by any means without the written permission of MeshNetics.

Disclaimer

MeshNetics believes that all information is correct and accurate at the time of issue. MeshNetics reserves the right to make changes to this product without prior notice. Please visit MeshNetics website for the latest available version.

MeshNetics does not assume any responsibility for the use of the described product or convey any license under its patent rights.

MeshNetics warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with MeshNetics standard warranty. Testing and other quality control techniques are used to the extent MeshNetics deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

Trademarks

MeshNetics®, ZigBit, ZigBeeNet, SensiLink, as well as MeshNetics and ZigBit logos are trademarks of MeshNetics Ltd.

All other product names, trade names, trademarks, logos or service names are the property of their respective owners.

Technical Support

Technical support is provided by MeshNetics.

E-mail: support@meshnetics.com

Please refer to Support Terms and Conditions for full details.

Contact Information

MeshNetics

EMEA Office
Am Brauhaus 12
01099, Dresden, Germany
Tel: +49 351 8134 228
Office hours: 8:00am - 5:00pm (Central European Time)
Fax: +49 351 8134 200

US Office
5110 N. 44th St., Suite L200
Phoenix, AZ 85018 USA
Tel: (602) 343-8244
Office hours: 9:00am - 6:00pm (Mountain Standard Time)
Fax: (602) 343-8245

Russia Office
9 Dmitrovskoye Shosse, Moscow 127434, Russia
Tel: +7 (495) 725 8125
Office hours: 8:00am - 5:00pm (Central European Time)
Fax: +7 (495) 725 8116

E-mail: info@meshnetics.com

www.meshnetics.com

Table of Contents

1. Intended Audience and Purpose 6

2. Introduction 7

2.1. Key Features 7

2.2. Supported Platforms 7

3. ZigBeeNet Architecture 8

3.1. Architecture Highlights 8

3.2. Naming Conventions 9

3.3. File System Layout 9

4. User Applications Programming 12

4.1. Event-driven programming 12

4.2. Request/confirm and indication mechanism 12

4.3. Task scheduler, priorities, and preemption 14

4.4. Application timers 15

4.5. Concurrency and interrupts 15

4.6. Typical application structure 17

5. Memory and resource allocation 20

5.1. RAM 20

5.2. Flash storage 22

5.3. EEPROM 22

5.4. Other resources 22

6. Application walk-through 24

Related Documents 34

© 2008 MeshNetics

List of Figures

Figure 1: Software Stack Architecture.................................. 8

Figure 2: Asynchronous calls vs. Synchronous calls.......... 12

Figure 3. Normal and interrupt control flow in the stack...... 16

© 2008 MeshNetics

List of Tables

Table 1: ZigBeeNet SDK file system layout 11

Table 2: Configuration server parameters and RAM consumption 20

Table 3: Hardware resources reserved for use by the stack22

© 2008 MeshNetics

1. Intended Audience and Purpose

The purpose of this document is to familiarize the audience of embedded software developers and system designers with ZigBeeNet SDK. The document covers the following topics:

1. ZigBeeNet stack architecture
2. User application programming model
3. Memory and resource allocation
4. System design considerations
5. Walk-through of key APIs with code samples and commentary

The audience is assumed to be familiar with the C programming language. Some knowledge of embedded systems is recommended but not required. The audience is assumed to be familiar with key aspects for ZigBee and ZigBee PRO standards for low-power wireless networking ([1]). You may refer to online tutorials at www.meshnetics.com/zigbee-tutorial/ for more information on ZigBee fundamentals.

2. Introduction

ZigBeeNet is a full-featured, next generation embedded software stack from MeshNetics. The stack provides a software development platform for reliable, scalable, and secure wireless applications running on MeshNetics ZigBit modules. ZigBeeNet is designed to support a broad ecosystem of user-designed applications addressing diverse requirements and enabling a full spectrum of software customization. Primary application domains include home automation, commercial building automation, automated meter reading, asset tracking, and industrial automation.

ZigBeeNet is fully compliant with ZigBee PRO and ZigBee standards for wireless sensing and control. It provides an augmented set of APIs which, while maintaining 100% compliance with the standard, offer extended functionality designed with developer's convenience and ease-of-use in mind. As seasoned ZigBee technology experts, we at MeshNetics created ZigBeeNet to dramatically lower the developer learning curve, factor out the unnecessary complexity and expose as much power of the underlying ZigBit hardware platform as possible. The stack incorporates three years worth of wireless system design experience, field work, and actual user feedback.

ZigBeeNet's target audience is system designers, embedded programmers and hardware engineers evaluating, prototyping, and deploying wireless solutions and products built around the ZigBit hardware platform. ZigBeeNet is delivered as a software development kit, which includes (1) extensive documentation, (2) standard set of libraries comprising multiple components of the stack, (3) sample user applications in source code, as well as (4) a complete set of peripheral drivers (also in source code) for the supported platforms.

2.1. Key Features

- Full ZigBee PRO and ZigBee compliance
- Easy-to-use C API and serial AT commands available
- Ultimate in data reliability with true mesh routing
- Large network support (100s of devices)
- Optimized for ultra low power consumption (5-15 years battery life)
- Extensive security API
- Over-the-air software update capability
- Flexible and easy to use developer tools

2.2. Supported Platforms

ZigBeeNet supports the following hardware platforms:

- ZDM-A1281-A2: ZigBit Module w/ dual chip antenna (HAL)
- ZDM-A1281-B0: ZigBit Module w/ balanced RF output (HAL)
- ZDM-A1281-PN: ZigBit Power Amplified Module w/ U.FL antenna connector (HAL)
- ZDM-A1281-PN0: ZigBit Power Amplified Module w/ unbalanced RF output (HAL)
- WDB-A1281-*: MeshBean2 Development Boards (BSP)

© 2008 MeshNetics

3. ZigBeeNet Architecture

3.1. Architecture Highlights

ZigBeeNet internal architecture follows 802.15.4, ZigBee-defined separation of the networking stack into logical layers. Besides the core stack containing protocol implementation, ZigBeeNet contains additional layers implementing shared services (e.g. task manager, security, and power manager) and hardware abstractions (e.g. hardware abstraction layer (HAL) and board support package (BSP)). The APIs contributed by these layers are outside the scope of core stack functionality. However, these essential additions to the set of APIs significantly help reduce application complexity and simplify integration. ZigBeeNet API reference manual provides detailed information on all public APIs and their use [7].

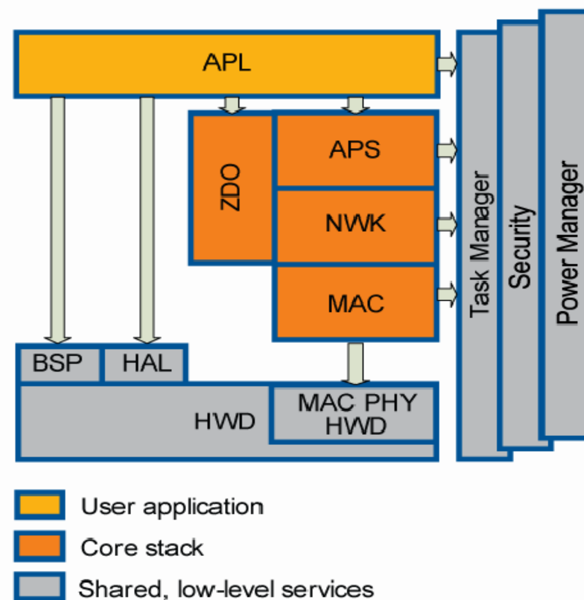

Figure 1: Software Stack Architecture

The topmost of the core stack layers, APS, provides the highest level of networking-related APIs visible to the application. ZDO provides a set of fully compliant ZigBee Device Object APIs which enable main network management functionality (start, reset, formation, join). ZDO also defines ZigBee Device Profile types, device and service discovery commands implemented by the stack.

There are three service "planes" including: task manager, security, and power manager. These services are available to the user application, and may also be utilized by lower stack layers. Task manager is the stack scheduler which mediates the use of the MCU among internal stack components and user application. The task manager utilizes a proprietary priority queue-based algorithm specifically tuned for multi-layer stack environment and demands of time-critical network protocols. Section 4.3 describes task scheduler and its interface in more detail. Power management routines are responsible for gracefully shutting down all stack components and saving system state when preparing to sleep and restoring system state when waking up.

© 2008 MeshNetics

Hardware Abstraction Layer (HAL) includes a complete set of APIs for using on-module hardware resources (EEPROM, app, sleep, and watchdog timers) as well as the reference drivers for rapid design-in and smooth integration with a range of external peripherals (IRQ, I2C, SPI, UART, 1-wire). Board Support Package (BSP) includes a complete set of drivers for managing standard peripherals (sensors, UID chip, sliders, and buttons) placed on a MeshBean development board.

3.2. Naming Conventions

Due to a high number of API functions exposed to the user, some 105 by the latest count, and many more structures and type definitions, a simple naming convention is employed to simplify the task of getting around and understanding user applications. Here are the basic rules:

1. Each API function name is prefixed by the name of the layer where the function resides. For example, `ZDO_GetLqiRssi` API is contributed by the ZDO layer of the stack.
2. Each function name prefix is followed by an underscore, '_', separating the prefix from the descriptive function name.
3. The descriptive function name may have a `'Get'` or `'Set'` prefix, indicating requesting that some parameter is returned or setting that parameter in the underlying stack, respectively (e.g. `HAL_GetSystemTime`)
4. The descriptive function name may have a `'Req'`, `'Request'`, `'Ind'`, or `'Conf'` suffix, indicating the following:

   '`Req`' and `'Request'` correspond to the asynchronous (see Section 4.2) *requests* from the user application to the stack (e.g. `APS_DataReq`)

   `'Ind'` corresponds to the asynchronous *indication* or events propagated to the user application from the stack (e.g. `ZDO_NetworkLostInd`)

   By convention, function names ending in `'Conf'` are the user-defined callbacks for the asynchronous requests, which *confirm* the request's execution.
5. Each structure and type name carries a '`_t`' suffix, standing for *type.*
6. Enumeration and #defined variable names are in ALL CAPS.

It is recommended that the application developer adhere to the aforementioned naming conventions in the user application.

3.3. File System Layout

The file system layout mirrors the stack architecture, with extra folders constituting sub-layers added. Those components distributed in binary form include `'lib'` and `'include'` directories, describing the component interface header files and containing the library image respectively. Those components distributed in source code include `'objs'` and `'src'` subdirectories, and a Makefile containing the build script which builds that component ([1], [2]).

The applications are always provided in source code and include the header and source file, the Makefile build script as well as project files for supported IDEs. The main action of the application Makefile is to compile the application using the header files found under

© 2008 MeshNetics

Component directories and to link the resulting object file with library images (also found under Components directory). The resulting binary image is a cross compiled application which can then be installed and debugged on the target platform. Further information on setting up developer build environment and tool chain can be found in [4], [5] and [6].

© 2008 MeshNetics

Table 1: ZigBeeNet SDK file system layout

Folder	Source code present?	Description
`ZigBeeNet`		Stack Root
`Components`		
`APS`	no	Application support sub-layer
`BSP`	yes	Board support package (reference drivers for supported evaluation and development boards)
`ConfigServer`	yes	Generic parameter storage sub-layer
`HAL`	yes	Hardware abstraction layer (reference drivers for supported platforms)
`MAC_PHY`	no	Media access control and physical layers
`NWK`	no	Network layer
`PersistDataServer`	no	
`SystemEnvironment`	no	main function and task manager
`ZDO`	no	ZigBee Device Object sub-layer
`Applications`		
`Blink`	yes	Simple blinking LEDs example
`LowPower`	yes	Simple application demonstrating power management and data acquisition
`Peer2peer`	yes	Simple application transferring data between two nodes
`PingPong`	yes	Simple application demonstrating multi-hop routing
`ThroughputTest`	yes	Application measuring stack throughput
`ZBNDemo`	yes	Full-featured WSN application demonstrating data acquisition, security, and power management

© 2008 MeshNetics

4. User Applications Programming

4.1. Event-driven programming

Event-driven systems are a common programming paradigm for small-footprint embedded systems with significant memory constraints and little room for the overhead of a full operating system. Event-driven or event-based programming refers to programming style and architectural organization which pairs each invocation of an API function with an asynchronous notification (and result of the operation) of the function completion is delivered through a callback associated with the initial request. Programmatically, the user application provides the underlying layers with a function pointer, which the layers below call when the request is serviced.

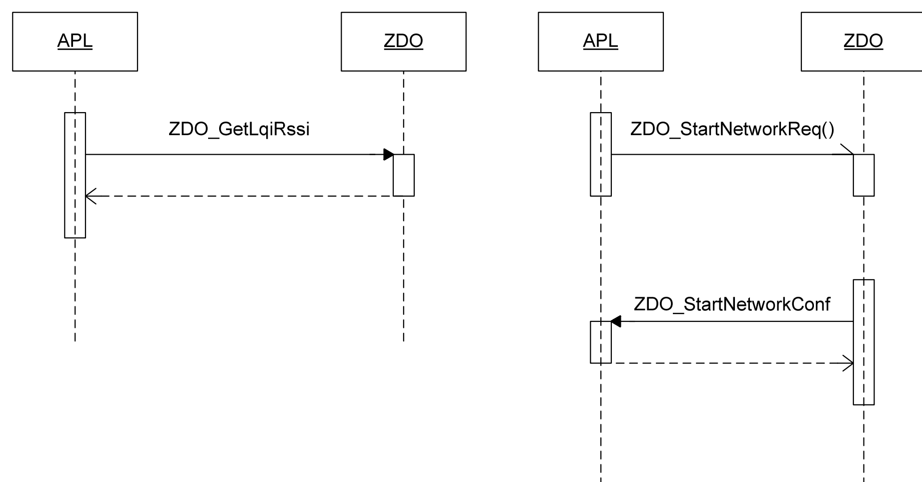

Figure 2: Asynchronous calls vs. Synchronous calls

Event-driven programming may also be familiar to embedded developers with GUI programming experience. GUI frameworks resort to the same callback mechanism when user-defined code needs to execute on some external system event (e.g. keyboard event or mouse click) providing the natural asynchrony. The similarity is superficial, however, as most user code interacting with a GUI framework is still synchronous, i.e. function calls block and the return value is typically retrieved immediately. In a fully event-driven system, all user code executes in a callback either a priori known to the system or registered with the stack by the user application. Thus, user application runs entirely in stack-invoked callbacks.

4.2. Request/confirm and indication mechanism

All application based on ZigBeeNet SDK are written in event-driven or event-based programming style. In fact, all internal stack interfaces are also defined in terms of downcalls and corresponding callbacks. Each layer defines a number of callbacks for the lower layers to invoke, and, in turn, invokes callback functions defined by higher levels. There is a generic type of user-defined callback which is responsible for executing application-level code called the task handler. `APL_TaskHandler` is the reserved

© 2008 MeshNetics

callback name known by the stack as the application task handler. Invocation of the APL_TaskHandler is discussed in Section 4.3.

Request/confirm mechanism is a particular instance of an event-driven programming model. Simply put, request is an asynchronous call to the underlying stack to perform some action on behalf of the user application; confirm is the callback that executes when that action completed and the result of that action is available.

For example, consider ZDO_StartNetworkReq(&networkParams) request call, which requests ZDO layer to start the network. The networkParams argument is a structure defined in zdo.h as

```
typedef struct
{
    uint32_t                              channelMask;
    uint64_t                              extPANId;

    ZDO_StartNetworkConf_t                confParams;
    void (*ZDO_StartNetworkConf)(ZDO_StartNetworkConf_t
*conf);
} ZDO_StartNetworkConf_t;
```

The first two fields of the structure store the requested channel mask and extended PAN ID. The third field is another structure used to the stack's response (actual network parameters) to the request. The last field is the actual callback pointer. The ZDO_StartNetworkReq(&networkParams) request is paired up with a user-defined function with the following signature:

```
static void ZDO_StartNetworkConf(ZDO_StartNetworkConf_t
*confirmInfo)
{

}
```

Not surprisingly, the request is preceded by an assignment to first, second, and forth fields of the structure, as follows:

```
networkParams.channelMask          = APP_CHANNEL_MASK;
networkParams.extPANId             = APP_COORD_UID;
networkParams.ZDO_StartNetworkConf = ZDO_StartNetworkConf;
```

The example illustrates a particular instance of using a request/confirm mechanism (see ZBNDemoApp.c in ZBNDemo folder for more detail), but all uses follow the same general setup.

Note that the need to decouple the request from the answer to that request is especially important when the request can take an unspecified amount of time. For instance, when requesting the stack to start the network, the underlying layers may perform an energy detecting scan which takes significantly longer than we are willing to block for. Sections 4.5 and 4.6 outline the reasons why prolonged blocking calls are not acceptable. For calls with deterministic execution times, synchronous calls are used. Calls from one user-defined function to another are vacuously synchronous. Some system calls, especially those with predictable execution times are also synchronous.

Apart from request/confirm pairs, there are cases when the application needs to be notified of an external event which is not a reply to any specific request. For this, there is a number of user-defined callbacks with fixed names which are invoked by the stack asynchronously. These include events indicating loss of network, readiness of the underlying stack to sleep, or notifying that the system is now awake.

© 2008 MeshNetics

System rule 1: All user applications are organized as a set of callbacks executing on completion of some request to the underlying layer.

System rule 2: User application is responsible for declaring callbacks to handle unsolicited system events of interest.

4.3. Task scheduler, priorities, and preemption

A major aspect of application development is managing the control flow and ensuring that different parts of the application do not interfere with each other's execution. In non-embedded applications, mediation of access to system resources is typically managed by the operating system which coordinates, for instance, that every application receives its fair share of system resources. Because multiple concurrent applications can coexist in the same system (also known as multitasking), the commodity operating system core like Windows are typically very complex in comparison to single-task systems like ZigBeeNet. In ZigBeeNet context, there is a single application running on top of the stack, thus most of contention for system resources happens not among concurrent applications but between the single application and the underlying stack. Both the stack and the application must execute their code on the same MCU.

In contrast to preemptive operating systems which are better suited to handle multiple applications but require significant overhead themselves, cooperative multitasking systems are low in overhead, but require, not surprisingly, cooperation of the application and the stack. Preemptive operating systems timeslice among different applications (multiplexing them transparently on one CPU) so that the application developers can have the illusion that their application has the exclusive control of the CPU. An application running in a cooperative multitasking system must be actively aware of the need to yield the resources that it's using (primarily, the processor) to the underlying stack.

Returning to the example with user callbacks, if `ZDO_StartNetworkConf` callback takes too long to execute, the rest of the stack will be effectively stalled waiting for the callback to return control to the underlying layer. Note that callbacks run under the priority of the invoking layer, so `ZDO_StartNetworkConf` runs under ZDO's priority level. Users should exercise caution when executing long sequences of instructions, including instructions in nested function calls, in the scope of a callback invoked from another layer.

System rule 3: All user callbacks should execute in 10 ms or less.

System rule 4: Callbacks run under the priority level of the invoking layer.

The strategy for callbacks executing longer than 10 ms is to defer execution. Deferred execution is a strategy to breaking up the execution context between the callback and the layer's task handler by using the task manager API. The way deferred execution is achieved is by preserving the current application state, and posting a task to the task queue as follows:

```
postTask(APL_TASK_ID);
```

Posting operation is synchronous, and the effect of the call is to notify the scheduler that the posting layer has a deferred task to execute. For the user application, the posting layer is always identified by APL_TASK_ID. Posting a task results in a deferred call to the application task handler, `APL_TaskHandler`, which, unlike other callbacks, runs under the application's priority level. In other words, the application task handler runs only when all higher priority tasks have completed. This permits longer execution time in a task handler's context versus a callback context.

System rule 5: Application task handler runs only when all tasks of higher priority have completed.

System rule 6: The application task handler should execute in 50 ms or less.

© 2008 MeshNetics

Additional task IDs of interest are HAL_TASK_ID and BSP_TASK_ID, which refer to tasks belonging to hardware abstraction layer or board support package, respectively. When a user application involves modifications to HAL or BSP layers, the deferred execution of HAL and BSP callbacks should utilize those layers' task IDs when posting.

4.4. Application timers

Thus far, the three ways that control flow enters application code are: (1) through task handler following a `postTask` invocation, (2) through confirm callbacks invoked by underlying stack on request completion, and (3) through asynchronous event notifications invoked by the stack. Note that neither one of the three has a time bound for when the invocation is to be expected. One way to ensure execution of a user-defined callback at a specific time in the future is by using a timer.

The stack provides a high-level application timer interface, which uses a low-level hardware timer. The timer interface is part of HAL and its use is illustrated in many sample applications (see `BlinkApp.c` in Blink folder for more detail). The general idea is that a `HAL_AppTimer_t` structure defines the timer interval (in milliseconds), whether the timer is a one shot timer or a timer firing continuously, and the callback function to invoke when the timer fires. The structure can then be passed to `HAL_StartAppTimer` and `HAL_StopAppTimer` to start and stop the timer, respectively.

4.5. Concurrency and interrupts

Concurrency refers to several independent "threads" of control executing at the same time. In preemptive systems with timeslicing and multiple threads of control, the execution of one function may be interrupted by the system scheduler at an arbitrary point, giving control to another thread of execution that could potentially execute a different function of the same application. Because of unpredictability of interruption and the fact that the two functions may share data, the application developer must ensure atomic access to all shared data.

As discussed previously, in ZigBeeNet a single thread of control is shared between the application and the stack. By running a task in a given layer of the stack, the thread acquires a certain priority, but its identity does not change--it simply executes a sequence of non-interleaved functions from different layers of the stack and the application. Thus the application control flow may be in no more than one user-defined callback at any given time. In the general case, the execution of multiple callbacks cannot be interleaved; each callback executes as an atomic code
block.

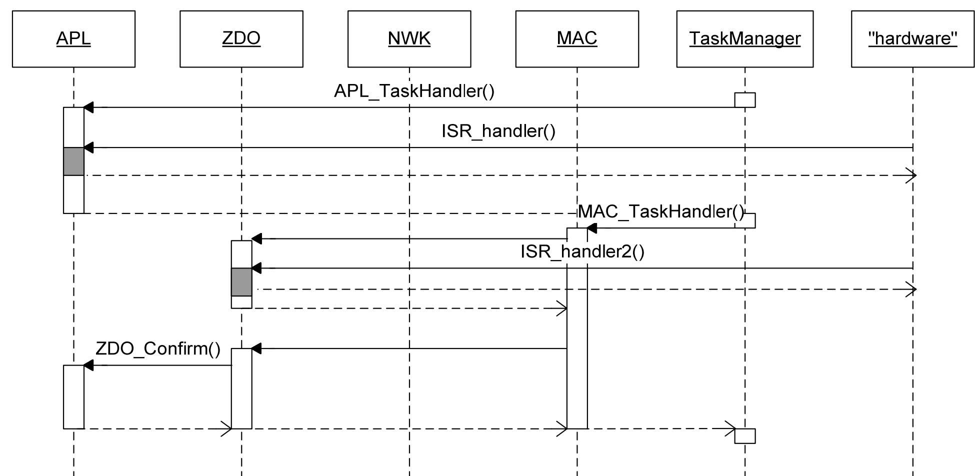

Figure 3. Normal and interrupt control flow in the stack

Even though timeslicing is not an issue, there is a special condition where another concurrent thread of control may emerge. This happens due to hardware interrupts, which can interrupt execution at an arbitrary point in time (the main thread of control may be either in the stack code or the application code when this happens) to handle a physical event from an MCU's peripheral device (e.g. UART or SPI channel). This is analogous to handling hardware interrupts in any other system.

Figure 3 illustrates an example interaction between the application, the stack, the task manager, and the hardware interrupts. Initially, the task handler processes an application task by invoking `APL_TaskHandler`. While the application-level task handler is running, it is interrupted by a hardware event (shown in gray). A function that executes on a hardware interrupt is called interrupt service routine or interrupt handler. After the interrupt handler completes, the control is returned to the application task handler. Once the task handler finishes, the control is returned to the scheduler, which selects a MAC layer task to run next. While the `MAC_TaskHandler` is running, it invokes a confirm callback in ZDO layer, and this callback is, in turn, interrupted by another hardware interrupt. Note also that the MAC task handler invokes another ZDO callback, which invokes another callback registered by the application. Thus, the application callback executes as if it had the priority of the MAC layer or `MAC_TASK_ID`.

A ZigBeeNet application may register an interrupt service routine which will execute on a hardware interrupt. Typically this is done by handling additional peripheral devices whose hardware drivers are not part of the standard SDK delivery and are instead added by the user. The call to add a user-defined interrupt handler is:

```
HAL_RegisterIrq(uint8_t irqNumber, HAL_irqMode_t irqMode,
void(*)(void) f);
```

The `irqNumber` is an identifier corresponding to one of the available hardware IRQ lines, `irqMode` specifies when the hardware interrupts are to be triggered, and `f` is a user-defined callback function which is the interrupt handler. Naturally, the execution of an interrupt handler may be arbitrarily interleaved with an execution of another application callback. If the interrupt handler accesses global state also accessed by any of the application callbacks than access to that share state must be made atomic. Failure to provide atomic access can lead to data races, i.e. non-deterministic code interleavings which will surely result in incorrect application behavior.

© 2008 MeshNetics

Atomic access is ensured by framing each individual access site with atomic macros `ATOMIC_SECTION_ENTER` and `ATOMIC_SECTION_LEAVE`. These macros start and end what's called a critical section, a block of code that is uninterruptible. The macros operate by turning off the hardware interrupts. The critical sections must be kept as short as possible to reduce the amount of time hardware interrupts are masked. Interrupts arriving during the masking are lost.

System rule 7: Critical sections should not exceed 50 μs in duration.

4.6. Typical application structure

A ZigBeeNet application differs significantly in its organization from a typical non-embedded C program. As discussed previously,

1. Every application defines a single task handler, which contains in its scope the bulk of the application's code (including code accessible through nested function calls)
2. Every application defines a number of callback functions contributing code which executes when an asynchronous request to the underlying layer is serviced.
3. Every application defines a number of callback with known names which execute when an event is processed by the stack.
4. Every application maintains global state that is shared state information between the callbacks and the task handler.

Notably, there is no `main()` function. Instead, the main function is embedded in the stack itself. Upon the stack initialization, the control passes from main to the task scheduler which begins invoking task handlers in order of priority (from highest to lowest) eventually invoking `APL_TaskHandler`, which is the initial entry point into the application. Following the initial call to the application task handler, the control flow passes between the stack and the callbacks as shown in Figure 3.

Application code may be arbitrarily split up among several C files (see ZBNDemo application and Makefile as an example). For simplicity's sake, we consider here a standard single file application omitting large portions of the code for illustration purposes.

```
/********************************
  #include directives
********************************/
...
#include <taskManager.h>
#include <zdo.h>
#include <configServer.h>
#include <aps.h>

/********************************
 FUNCTION PROTOTYPES
********************************/
...

/********************************
  GLOBAL VARIABLES
********************************/
AppState_t appState = APP_INITING_STATE;
...

/********************************
```

```
 IMPLEMENTATION
******************************/

/******************************
 Application task handler
******************************/
void APL_TaskHandler()
{
    switch (appState) {
        case APP_IN_NETWORK_STATE:
            ...
            break;
        case APP_INITING_STATE: //node has initial state
            ...
            break;
        case APP_STARTING_NETWORK_STATE:
            ...
            break;
    }
}

/******************************
 Confirm callbacks
******************************/
static void ZDO_StartNetworkConf(ZDO_StartNetworkConf_t
*confirmInfo)
{
    ...
    if (ZDO_SUCCESS_STATUS == confirmInfo->status) {
        appState = APP_IN_NETWORK_STATE;
        ...
        postTask(APL_TASK_ID);
    }
}

void ZDP_LeaveResp(ZDP_ResponseData_t *zdpRsp)
{
    ...
}

/******************************
 Indication callbacks
******************************/
void ZDO_NetworkLostInd(ZDO_NetworkLostInd_t *indParams)
{
    ...
    if (APP_IN_NETWORK_STATE == appState) {
        appState = APP_STARTING_NETWORK_STATE;
        ...
        postTask(APL_TASK_ID);
    }

}

void ZDO_MgmtNwkUpdateNotf(ZDO_MgmtNwkUpdateNotf_t
*nwkParams)
{
    ...
}
```

© 2008 MeshNetics

The above skeleton code is representative of the vast majority of user applications. Most applications are never more than several hundred lines of code (for instance, our most complex application, ZBNDemo, is some 1050 lines spread over 7 source files). Since network management and data transmission code is actually quite universal, the main differences among applications is in managing the peripherals and parsing application-level protocols.

Invariably, there is a global state, represented in our case by `appState` variable that is accessed by callbacks and the task handler. In real-world applications, state is represented by a number of variables for device role-dependent sub-states, various network parameters, sensor state, etc. Note that a typical task handler switches on a state variable to execute the code for each particular state, which is a programming template shared by all sample applications provided with the SDK. Application developer is well-served by considering their application first in terms of a state machine. The mapping from a state machine description to code is then natural given the event-driven programming style.

Note also the use of `postTask` scheduler function. For example, the application purposefully defers processing of a network lost indication to the task handler, where it is dealt with fully. The callback simply changes the global state and returns to the ZDO layer from which it is invoked. This style of programming is consistent with cooperative multitasking system setup, and it permits the stack to handle higher priority tasks before the returning to the deferred action.

© 2008 MeshNetics

5. Memory and resource allocation

5.1. RAM

RAM is a critical system resource used by ZigBeeNet to store runtime parameters like neighbor tables, routing tables, children tables, etc. As sometimes required by the stack, the values of certain parameters stored in EEPROM are cached in RAM for easier subsequent retrieval. The call stack also resides in RAM, and is shared by the stack and the user application. To conserve RAM, the user must refrain from use of recursive functions, functions taking many parameters, functions which declare large local variables and arrays, or functions that pass large parameters on the stack.

System rule 8: Whole structures should never be passed on the stack, i.e. used as function arguments. Use structure pointers instead.

System rule 9: Global variables should be used in place of local variables, where possible.

User defined callbacks already ensure that structures are passed by pointer. The user must verify that the same is true for local user-defined functions taking structure type arguments.

System rule 10: The user application must not use recursive functions. It is recommended that the maximum possible depth of nested function calls invoked from a user-defined callback is limited to 10, each function having no more than 2 pointer parameters each.

Overall, the RAM demands of the stack must be reconciled with that of the user application. Fortunately, the amount of RAM used by the stack data is a user-configurable parameter. The configuration server, ConfigServer or CS, is distributed in source code (see Table 1), to allow the user to redefine the key runtime parameters of the stack. Among these parameters are parameters directly affecting the size of runtime tables stored in RAM. The table below lists such parameters and provides an easy formula to compute RAM consumption based on the value of a parameter (p):

Table 2: Configuration server parameters and RAM consumption

Parameter name	Description	RAM consumption
CS_NEIB_TABLE_SIZE	Number of entries in neighbor table	51 * p
CS_ROUTE_TABLE_SIZE	Number of entries in routing table	6 * p
CS_DUPLICATE_REJECTION_TABLE_SIZE	Number of entries in duplicates filtering table	7 * p
CS_APS_GROUP_TABLE_ENDPOINTS_AMOUNT	Number of endpoints in a group	(3 + CS_APS_GROUP_TABLE_ENDPOINTS_AMOUNT) * CS_APS_GROUP_TABLE_GROUPS_AMOUNT
CS_APS_GROUP_TABLE_GROUPS_AMOUNT	Number of groups	

© 2008 MeshNetics

Parameter name	Description	RAM consumption
CS_ADDRESS_MAP_TABLE_SIZE	Number of entries in address map table	**11 * p**
CS_ROUTE_DISCOVERY_TABLE_SIZE	Number of entries in temporary table for route discovery	**12 * p**
CS_NWK_DATA_REQ_BUFFER_SIZE	Number of entries in NWK data requests buffer	**90 * p**
CS_NWK_DATA_IND_BUFFER_SIZE	Number of entries in NWK data indications buffer	**165 * p**
CS_APS_DATA_REQ_BUFFER_SIZE	Number of entries in APS data requests buffer	**34 * p**
CS_APS_ACK_FRAME_BUFFER_SIZE	Number of entries in APS acknowledgements buffer	**76 * p**

The overall contribution to the RAM consumption by the stack can be computed by summing up the last column of the table and substituting the value of each parameter for the respective p. With reasonable values of CS parameters, the user can expect about 5-6K of RAM to be consumed by the data allocated by the stack. The remainder of RAM is accessible to the user application data *and the call stack.* In addition to the user-tunable CS parameters, there is a fixed RAM allocation dedicated to the stack.

At compile time, RAM consumed by the data stored in RAM may be determined by running the following command. This command is usually appended to the Makefile of all provided sample applications:

```
avr-size -d app.elf
```

The result will be presented as below:

text	data	bss	dec	hex	filename
118482	714	6068	125264	1e950	app.elf

The number of bytes consumed by data will be **data + bss**. Note that this value does not include the portion of RAM used by the call stack, which varies at runtime and depends on the depth of the stack and the number of function parameters. The user may inspect the value of the stack pointer (pointing to the top of the stack) at runtime by running the application in debug mode. An important property of the system is that all memory allocation is static, i.e. the amount of RAM consumed by the stack and the user application *data* is known at compile time.

Most C programmers are familiar with C lib functions like malloc(), calloc(), realloc(), which are used to allocate a block of RAM at runtime. The use of these functions is strictly prohibited, even though they may be accessible from the avr-libc library linked with the stack and applications. Because dynamic allocation imposes additional memory management overhead, expressed in both delay in handling application requests for memory and bookkeeping overhead of tracking each available

© 2008 MeshNetics

memory block, its use is considered infeasible when dealing with 2-3K of RAM available to an application.

System rule 11: Dynamic memory allocation is strictly prohibited. The user must refrain from using standard `malloc()`, `calloc()`, and `realloc()` C lib function calls.

5.2. Flash storage

Another critical system resource is flash memory. The embedded microcontroller uses the flash memory to store program code. The footprint of the application in flash may be determined by running the following command:

```
avr-size -d app.elf
```

The result will be presented as below:

```
text      data      bss       dec       hex       filename
118482    714       6068      125264    1e950     app.elf
```

The number of bytes consumed by will be **`text + data`.** Unlike with RAM, the user has little control of how much flash space is consumed by the underlying stack. Since the stack libraries are delivered as binary object files, at link time (part of application building process) the linker ensures that mutual dependencies are satisfied, i.e. that the API calls used by the application are present in the resulting image, and that the user callbacks invoked by the stack are present in the user application. The linking process does not significantly alter the amount of flash consumed by the libraries.

5.3. EEPROM

EEPROM constitutes non-volatile storage available on most microcontrollers. Since EEPROM is another resource shared by both the stack and the application to store its non-volatile data, the use of EEPROM is arbitrated by a special API called Persistence Data Server or PDS ([7]). In the general case, EEPROM has linear addresses, so in order to protect the EEPROM portion occupied by stack parameters from being written over by the application code, PDS uses a special offset to write and read all application data.

PDS also detects any CRC errors by automatically storing a checksum alongside every parameter stored in EEPROM. When a parameter is read, the checksum is computed and compared to the one stored in EEPROM.

5.4. Other resources

Additional hardware resources include microcontroller peripherals, buses, timers, IRQ lines, I/O registers, etc. Since many of these interfaces have corresponding APIs in hardware abstraction layer (HAL), the user is encouraged to use the high-level APIs instead of the low-level register interfaces to ensure that the resource use does not overlap with that of the stack. The hardware resources reserved for the internal use by the stack are listed below.

Table 3: Hardware resources reserved for use by the stack

Resource	Description
Processor main clock	4 or 8 MHz from internal RC-oscillator or external radio frequency

© 2008 MeshNetics

Resource	Description
SPI	Radio interface
ATmega ports PB0, PB1, PB2, PB3, PB4, PA7, PE5	Radio interface
ATmega port PC1	Interface for amplifier (if present)
ATmega ports PG3, PG4	Asynchronous timer interface
Timer/Counter1	Radio interface
Timer/Counter2	Asynchronous timer
Timer/Counter4	System timer
External IRQ4	For wake-up on DTR command
External IRQ5	Radio interface
EEPROM	Storage for user settings accessible via Persist Data Server.

System rule 12: Hardware resources reserved for use by the stack must not be accessed by the application code.

© 2008 MeshNetics

6. Application walk-through

In this section an example of a working application (named lowpower) is described in detail. Coordinator part of lowpower sample application as implemented in *coordinator.c* file is reviewed here, together with l*owpower.h* header file with application specific types, variables and constants. Files are split into several logical blocks with functional description and comments provided next to each block.

lowpower.h contains definitions common for all components (coordinator and end device) in the lowpower sample application.

```
/**************************************************************************
  Lowpower.h
  Copyrights Meshnetics
***************************************************************************/
#include <configServer.h>
…
#include <appTimer.h>

#ifndef TIMER_STARTING_NETWORK
#define TIMER_STARTING_NETWORK 500 // LED blinking interval during network
                                   // start, ms
#endif
```

In order to access specific API functionality corresponding header files are included in the beginning of the file. If the constant *TIMER_STARTING_NETWORK* has been defined in Makefile, it will not be modified. Otherwise, the default value of 500 ms is assigned.

```
#define NETWORK_BUTTON   1  // Button that starts a network
#define NETWORK_LED LED_RED // Network indication red LED
#define NETWORK_TRANSMISSION_LED LED_YELLOW // Data transmission yellow LED
#define COORDINATOR_NETWORK_ADDRESS 0    // Coordinator NWK addres; must be 0
#define END_POINT   2        // End-point for transmission and receive
```

Constants that are not configurable via makefile are defined above.

```
typedef enum                // Application states
{
  APP_INITING_STATE,
  APP_AWAITING_NETWORK_START_STATE,
  APP_NETWORK_STARTING_STATE,
  APP_NETWORK_STARTING_STATE,
  APP_IN_NETWORK_STATE,
  APP_NO_NETWORK_STATE
} AppState_t;
```

Application state enum is defined in code above. Its usage is explained below.

```
typedef struct                  // Message structure on application level
{
  ShortAddr_t shortAddr;        // Source NWK address, not used
  uint32_t temperature;         // Temperature value
} PACK AppMessage_t;

typedef struct                  // Transmission buffer construction
{
  uint8_t         header[APS_ASDU_OFFSET];
  AppMessage_t    data;
  uint8_t         footer[APS_AFFIX_LENGTH - APS_ASDU_OFFSET];
} PACK AppMessageRequest_t;
```

Structures *AppMessage_t* and *AppMessageRequest_t* are used to transfer data from end device to coordinator.

```
/***************************************************************************
  Global variables
***************************************************************************/
AppState_t     appState = APP_INITING_STATE; // Current application state
extern DeviceType_t appDeviceType; // Device type (Coordinator/Router/End
                                   // device);
extern uint64_t        appUid;       // Unique ID of the device
// eof lowpower.h
```

Variables used by both end device and coordinator are also defined in *lowpower.h* file. Variable *appState* is initialized equally for both node types while *appDeviceType* and *appUid* will be initialized differently for end device and for coordinator nodes and hence declared as extern(al).

Functionality specific to coordinator in lowpower sample application is implemented in *coordinator.c* file described below.

```
/**************************************************************************
  coordinator.c
  Copyrights Meshnetics
  ZigBeeNet Low-Power: Coordinator part of application implementation.
**************************************************************************/
#include "lowpower.h"
#include <uart.h>
#include <apsDataManager.h>
```

coordinator.c file starts with *#include* section. Variables and functions defined in *lowpower.h* file are used by coordinator code so this file needs to be included. In addition some functionality required only for coordinator (in terms of lowpower sample application) is enabled by including *uart.h* and *apsDataManager.h* files.

```
/***************************************************************************
  Global variables
***************************************************************************/
```

```
DeviceType_t appDeviceType = DEVICE_TYPE_COORDINATOR; // Device type (should
not be changed)
uint64_t appUid = 0;  // Unique ID of the device (will be changed)
```

Global external variables which are declared in *lowpower.h* are initialized next. As device type is already known, it is set to *DEVICE_TYPE_COORDINATOR*. Unique device ID (*appUid* used as MAC address) is subject to change, so it is initialized with default value 0 that will be overwritten later.

```
/*************************************************************************
  Local variables
*************************************************************************/
static ZDO_StartNetworkReq_t  networkParams; // Request parameters for
                                             // ZDO_StartNetworkReq
static HAL_AppTimer_t startingNetworkTimer; // Timer indicating network start
// Endpoint parameters
static SimpleDescriptor_t simpleDescriptor = {END_POINT, 1, 1, 1, 0, 0, NULL,
0, NULL};
static APS_RegisterEndpointReq_t endpointParams;

static uint8_t tmpBuf[40];    // Output buffer for transmission to UART
static bool uartBusy = false; // Indicates whether UART is busy
static HAL_UartDescriptor_t appUartDescriptor; // UART descriptor
static uint8_t writePending = 0;
```

Local variables accessible only in *coordinator.c* file are defined and some of them are also initialized in the code block displayed above. These variables are declared as static in order to make them always accessible during the application run-time.

Most common ZDO requests (*ZDO_StartNetworkReq_t, ZDO_WakeUpReq_t, etc.*) that can be initialized in one function and used in different parts of the file are defined as local variables. Same is valid for application specific timers, end point and UART descriptors, and so on.

```
/*************************************************************************
  Local functions
 ************************************************************************/
// Network start/join confirmation handler
static void ZDO_StartNetworkConf(ZDO_StartNetworkConf_t *confInfo);
static void APS_DataInd(APS_DataInd_t *indData); // Data reception handler
static void appInitUARTManager(); // Initial UART configuration
static void appWriteDataToUart(uint8_t* aData, uint8_t aLength); // Data
transmission to UART
static void appUartWriteConfirm(); // Confirmation of UART write operation
```

Local functions accessible only in *coordinator.c* are declared next. None of the functions is expected to be available outside of *coordinator.c* file so they are defined as static. As it can be seen from the comments inside the code callbacks for network start confirmation, data indication, writing to UART as well as UART initializing and writing functions are defined in *coordinator.c.* Note that functions that are defined inside the ZigBeeNet stack but require implementation in user application shall not be redefined in the application file (example *APL_TaskHandler()*, *ZDO_MgmtNwkUpdateNotf()*, *ZDO_SleepInd*, and so on).

After variables and functions are declared, the declared functions are implemented as shown below.

© 2008 MeshNetics

```
/***********************************************************************
  Implementation
***********************************************************************/

/***********************************************************************
  Application task handler
  Parameters:
    None
  Return:
    None
***********************************************************************/
void APL_TaskHandler()
{
  switch (appState)
  {
    // Node is in initial state
    case APP_INITING_STATE:
          BSP_OpenLeds();
          uint8_t dswitch = BSP_ReadSliders(); // Read dipswitch's state.

          // Coordinator has DIP switches turned off
          if (dswitch == 0)
          {
                endpointParams.simpleDescriptor = &simpleDescriptor;

                // Set extended PANID
                uint64_t ext_pan = APP_COORD_UID;
                CS_WriteParameter(CS_EXT_PANID_ID, &ext_pan);

                // Assign own MAC address
                appUid = APP_COORD_UID;
                CS_WriteParameter(CS_UID_ID, &appUid);

                // Set transceiver to be always on (no sleeping)
                bool rxOnWhenIdleFlag = true;
                CS_WriteParameter(CS_RX_ON_WHEN_IDLE_ID, &rxOnWhenIdleFlag);
                // Set device Type
                CS_WriteParameter(CS_DEVICE_TYPE_ID, &appDeviceType);
                // Register handler for incoming data packets
                endpointParams.APS_DataInd = APS_DataInd;
                // Configure  static network addressing
                bool useStatAddr = true;
```

© 2008 MeshNetics

```
            CS_WriteParameter(CS_NWK_UNIQUE_ADDR_ID, &useStatAddr);
            uint16_t statAddr = COORDINATOR_NETWORK_ADDRESS;
            CS_WriteParameter(CS_NWK_ADDR_ID, &statAddr);
            appInitUARTManager(); // Initialize UART
            // Set network start request parameters
            networkParams.channelMask          = APP_CHANNEL_MASK;
            networkParams.extPANId             = APP_COORD_UID;
            // Handler for network start confirmation
            networkParams.ZDO_StartNetworkConf = ZDO_StartNetworkConf;
            appState = APP_AWAITING_NETWORK_START_STATE; // Switch state
        }
        SYS_PostTask(APL_TASK_ID);
        break;
```

The code block above shows the beginning of the *APL_TaskHandler* function. It is used to handle user application tasks as described in Section 4.3. *APL_TaskHandler* also acts as the entry point into the user application, because it is automatically executed by the task scheduler at least once. In order to be able to process different tasks an application state variable is used (*appState* as defined in *lowpower.h*). Tasks that correspond to different application states are processed in the corresponding *case* blocks inside the *switch(appState)* statement.

appState variable is initialized in *lowpower.h* file as *APP_INITING_STATE* and therefore after a node is powered up and the control is passed over to APL_TaskHandler() the code inside the *case APP_INITING_STATE* block is executed. That is why all node initialization procedures are done here (such as setting node type and node addresses). Since the starting network request parameters (channel mask and confirmation callback function) are not going to be modified during the network deployment, they are assigned here too. After initialization application state is changed to *APP_AWAITING_NETWORK_START_STATE* and hence *APL_TaskHandler* needs to be executed again. To achieve this, we use *SYS_PostTask(APL_TASK_ID)* in the end of the code block, thus putting a request to call the APL_TaskHandler in the scheduler queue.

```
    // Node is awaiting network start request
    case APP_AWAITING_NETWORK_START_STATE:
        // Set handlers for button events
        BSP_OpenButtons(NULL, buttonReleased);
        break;
```

Once application is in *APP_AWAITING_NETWORK_START_STATE* state, the buttons located on the MeshBean board are initialized with corresponding callback function for button release event and no callback (NULL) for button press event. The application state is not changed yet and hence there is no need to put an APL task in queue.

```
// Node starting a network
    case APP_NETWORK_STARTING_STATE:
         ZDO_StartNetworkReq(&networkParams); // Start network
         break;
```

In *APP_NETWORK_STARTING_STATE* block coordinator actually initiates network start using parameters set before.

```
    // Node is in network
    case APP_IN_NETWORK_STATE:
      break;
```

```
    // Node is out of network
    case APP_NO_NETWORK_STATE:
      break;

    default:
      break;
  }
}
```

For the application states *APP_IN_NETWORK_STATE* and *APP_NO_NETWORK_STATE* no commands are executed. These states are used as placeholders to show the possibilities for task processing when nodes are in (or out of) the network. However it does not mean that nothing happens on the application level. Network events are processed in other functions without changing the application state.

```
/**********************************************************************
  Starting network timer has fired. Toggle LED for blink
  Parameters:
    None
   Returns:
    None
**********************************************************************/
void startingNetworkTimerFired()
{
  BSP_ToggleLed(NETWORK_LED);
}
```

Event handler for network timer started after button release event. Used just as indication of network starting (blinking LED).

```
/******************************************************************************
  ZDO_StartNetwork primitive confirmation was received.
  Parameters:
    confirmInfo - confirmation information
  Return:
    None
******************************************************************************/
static void ZDO_StartNetworkConf(ZDO_StartNetworkConf_t *confInfo)
{
  static uint8_t msg[40] = "";  // Buffer for message to UART
  uint8_t strLength = 0;

  // Network start was successful
  if ((APP_NETWORK_STARTING_STATE == appState) &&
      (ZDO_SUCCESS_STATUS == confInfo->status))
  {
     HAL_StopAppTimer(&startingNetworkTimer); // Stop blinking
     BSP_OnLed(NETWORK_LED); // Indicate successful network start

     // Status message
```

```
      strLength = sprintf((char*)msg, "Network started...\r\n");

      appWriteDataToUart(msg, strLength); // Write status message to UART

      // Update stack profile and register application end point
      CS_WriteParameter(CS_STACK_PROFILE_ID, &endpointParams.simpleDescriptor-
>AppProfileId);
      APS_RegisterEndpointReq(&endpointParams);
      appState = APP_IN_NETWORK_STATE; // Switch state
   }
   else
   {
      strLength = sprintf((char*)msg, "Network start failed. Trying
again...\r\n"); // Status message
      appWriteDataToUart(msg, strLength); // Write status message to UART
   }
   SYS_PostTask(APL_TASK_ID);
}
```

In this function network start confirmation is processed and corresponding message is transmitted to UART. If the node was in the *APP_NETWORK_STARTING_STATE* and the confirmation status is ZDO_SUCCESS_STATUS then coordinator has successfully started the network. This is indicated by turning *NETWORK_LED* on and sending a *msg* to UART. An important step here is end point registration and application state update to *APP_IN_NETWORK_STATE*. Therefore *SYS_PostTask(APL_TASK_ID)* is executed afterwards.

If the network start was unsuccessful or coordinator is already in the network, failure message is transmitted to the UART.

```
/*****************************************************************************
  Handler of button release
  Parameters:
    buttonNumber - button number being released
  Return:
    None
*****************************************************************************/
void buttonReleased(uint8_t buttonNumber)
{
  if ((APP_AWAITING_NETWORK_START_STATE == appState) &&
      (NETWORK_BUTTON == buttonNumber))
  {
      // set and start timer for LED blinking during network start
      startingNetworkTimer.interval = TIMER_STARTING_NETWORK;
      startingNetworkTimer.mode     = TIMER_REPEAT_MODE;
      startingNetworkTimer.callback = startingNetworkTimerFired;
      HAL_StartAppTimer(&startingNetworkTimer);
      appState = APP_NETWORK_STARTING_STATE;
      SYS_PostTask(APL_TASK_ID);
  }
}
```

© 2008 MeshNetics

buttonReleased function is executed upon button release event as specified above in *BSP_OpenButtons(NULL, buttonReleased).* As you can see from the code, first it checks whether the node is awaiting network start and that release of the corresponding button (*NETWORK_BUTTON*) has triggered the callback. If both conditions are met, *startingNetworkTimer* is configured and started to indicate network start procedure. *appState* is changed and therefore APL task is posted to queue using *SYS_PostTask(APL_TASK_ID)*. Release of buttons other than *NETWORK_BUTTON* is not handled in this application.

```
/*******************************************************************
  Network update notification
  Parameters:
    ZDO_MgmtNwkUpdateNotf_t *nwkParams - update notification
  Returns:
    None
*******************************************************************/
void ZDO_MgmtNwkUpdateNotf(ZDO_MgmtNwkUpdateNotf_t *nwkParams)
{
  // Network is lost, initiate network rejoin procedure
  if (ZDO_NETWORK_LOST_STATUS == nwkParams->status)
  {
    // Start network join indication
    HAL_StartAppTimer(&startingNetworkTimer);

    appState = APP_NETWORK_STARTING_STATE;
    SYS_PostTask(APL_TASK_ID);
  }
}
```

Function shown above is a notification callback which is defined inside the ZDO library but has to be implemented in your application. In *coordinator.c* only "network lost" indication is handled. As you can see, if network is lost, network start timer is started again and application state is changed to *APP_NETWORK_STARTING_STATE* in order to restart the network.

```
/*******************************************************************
  Init UART, register UART callbacks.
  Parameters:
    None
  Returns:
    None
*******************************************************************/
void appInitUARTManager()
{
   appUartDescriptor.tty          = UART_CHANNEL_1; // UART channel
   appUartDescriptor.mode         = ASYNC;  // UART synchronization mode
   appUartDescriptor.baudrate     = UART_BAUDRATE_38400; // UART baud rate
```

```
    appUartDescriptor.dataLength   = UART_DATA8; // UART data length
    appUartDescriptor.parity       = UART_PARITY_NONE; // UART parity mode.
    appUartDescriptor.stopbits     = UART_STOPBIT_1; // UART stop bit
    appUartDescriptor.flowControl = UART_FLOW_CONTROL_NONE; // Flow control
    appUartDescriptor.rxBuffer     = NULL;
    appUartDescriptor.rxBufferLength   = 0;
    appUartDescriptor.txBuffer     = NULL;
    appUartDescriptor.txBufferLength   = 0;
    appUartDescriptor.txCallback   = appUartWriteConfirm; // Callback function,
                                             //confirming data transmission
  HAL_OpenUart(&appUartDescriptor);
}
```

appInitUARTManager function is executed when node is in *APP_INITING_STATE* in order to initialize UART port. Local variable *appUartDescriptor* acts as UART port identifier with all necessary configuration parameters. After calling *HAL_OpenUart(&appUartDescriptor)* function corresponding UART port is opened for use inside the application.

```
/*********************************************************************
  Write data to UART port.
  Parameters:
    uint8_t* aData  - pointer to data array
    uint8_t aLength - data array length
  Returns:
    None
*********************************************************************/
void appWriteDataToUart(uint8_t* aData, uint8_t aLength)
{
  // UART port is occupied
  if (uartBusy)
  {
     // Temporary store message in special buffer
     strncpy((char*)tmpBuf, (char*)aData,aLength);
     // Increase data size stored in the temporary buffer
     writePending += aLength;
  }
  // UART port is free
  else
  {
     uartBusy = true; // Mark UART port as occupied
     // Transmit data to UART
     HAL_WriteUart(&appUartDescriptor, (void*)aData, aLength);
  }
}
```

appWriteDataToUart function transmits data passed as an argument with specified length to UART port. In the Lowpower sample application single UART port is used, so UART descriptor is not passed to the function as an argument, but is taken from a global variable. A port occupancy check is done first and if UART is currently

© 2008 MeshNetics

busy, the data is stored in temporary buffer to be transmitted later. If UART is free then data is sent directly to UART using *HAL_WriteUart* function.

```
/*****************************************************************************
  Write confirmation has been received. New message can be sent.
  Parameters:
    None
  Return:
    None
*****************************************************************************/
static void appUartWriteConfirm()
{
  uartBusy = false; // UART port has been released

  // There is data waiting to be written to UART
  if (writePending)
  {
     // Write temporary stored data to UART
     appWriteDataToUart(tmpBuf, writePending);
     writePending = 0;
  }
}
```

appUartWriteConfirm function is a UART writing confirmation callback as specified before in UART configuration parameters. It is called when UART is ready to write more data. Additional data that might have arrived for transmission to UART can be sent now by *appWriteDataToUart*.

```
void ZDO_SleepInd()
{
}

void ZDO_WakeUpInd()
{
}
//eof coordinator.c
```

Functions above are declared in ZDO libraries and require implementation in user application. Because coordinator has no sleep functionality these functions are left empty.

© 2008 MeshNetics

Related Documents

[1] GNU 'make'. http://www.gnu.org/software/make/manual/make.html

[2] make (software). http://en.wikipedia.org/wiki/Make_(software)

[3] 053474r17ZB_TSC-ZigBee stack profile Pro Specification. http://www.zigbee.org/en/spec_download/download_request.asp

[4] Creating, building & debugging ZigBeeNet applications using AVR Studio. Application Note. Meshnetics Doc. AN-481~07

[5] Creating, building & debugging ZigBeeNet applications using Eclipse IDE. Application Note. Meshnetics Doc. AN-481~08

[6] ZigBit Development Kit Users Guide. Meshnetics Doc. S-ZDK-451

[7] ZigBeeNet Stack Documentation. Meshnetics Doc. P-ZBN-452~02

© 2008 MeshNetics