Development of a concept for Over The Air Programming of Sensor Nodes

Masterarbeit
Chemnitz, Januar 2016

Zur Erlangung des akademischen Grades
Master of Science

Fakultät Informatik
Professur Technische Informatik

Eingereicht von: Anantha Ramakrishna Jayaram
Geburtsort: Bangalore, Indien

Betreuer: Dipl.-Inf. Mirko Lippmann
Acknowledgment

The completion of this Master Thesis commemorates the realization of a long awaited dream of mine to see myself achieve this position in my life. Foremost, I would like to express my sincere gratitude to my advisor Dipl.-Inf. Mirko Lippmann for his patience, valuable feedbacks and, most of all, giving me the flexibility to work freely. I am ever so grateful to Prof. Dr. Wolfram Hardt and Technische Universität Chemnitz for providing me an opportunity to pursue my Master's degree. A heartfelt thanks to every personnel from Technische Universität Chemnitz who supported and guided me throughout my Master’s degree.

Reaching this milestone in the course of my life wouldn't be possible if not for the unwavering love and support from my family and God's grace. Hence I would like to give a special thanks to them. Finally, I would like to thank every single person in my life's journey so far. May it be because of the positive support that they have given me or for showing the negativity from which I learnt to improve myself by not following such traits.
Abstract

Nowadays, wireless sensor networks can be found in many new application areas. In these sensor networks there may exit a part of the network which are difficult to access or lie in a wide area, far apart. A change in the software (e.g., function update or bug fix) can entail reprogramming of all sensor nodes. This is very time consuming and labour intensive, if the patching has to be done manually for each individual sensor nodes.

In the area of mobile phones, the over the air (OTA) update function has been established very well with good reliability. In embedded systems such as sensor nodes, where resources are severely restricted, an update cannot be stored but must be programmed directly with the transfer. For this to be possible, a lot of basic functionality is needed to be established to correct errors or to be able to resume a failed programming.

Within the framework of this thesis a concept for the transmission and distribution of the firmware and programming the sensor node is established. Focus here is to optimize the use of resources and to provide basic functionality within the programming mode.
Table of Contents

1 INTRODUCTION ............................................................................................................. 10
  1.1 Overview .................................................................................................................. 10
  1.2 Report Organization ............................................................................................... 13

2 MOTIVATION AND OBJECTIVE .................................................................................. 14

3 STATE OF THE ART ................................................................................................... 15

4 CONCEPT DESCRIPTION ............................................................................................. 17
  4.1 Firmware Bifurcation .............................................................................................. 17
  4.2 OTA Updater type Bootloader ............................................................................... 19
  4.3 OTA Update Receiver type Bootloader .................................................................. 21
  4.4 Communication ...................................................................................................... 23
  4.5 Effects of Firmware Bifurcation ............................................................................. 23

5 IMPLEMENTATION ....................................................................................................... 25
  5.1 Hardware .................................................................................................................. 25
    5.1.1 Nanett Evaluation Board .................................................................................. 25
    5.1.2 MSP-FET430UIF USB Debugging Interface .................................................... 26
  5.2 Software ................................................................................................................... 27
    5.2.1 CrossStudio with CrossWorks for MSP430 ...................................................... 27
  5.3 Implementation of Bootloader State Machines ......................................................... 27
    5.3.1 OTA Updater type Bootloader State Machine .................................................. 27
    5.3.2 OTA Update Receiver type Bootloader State Machine .................................... 30
    5.3.3 Bootloader Memory Requirement .................................................................... 34
  5.4 OTA Network Packet Types and Structure ............................................................... 34
  5.5 OTA Network Packet Exchange Progression ........................................................... 39
  5.6 Configuration Parameters for OTA Updater type Bootloader .................................. 42
  5.7 Section Placement File ........................................................................................... 46
5.8 Interrupt Vector Redirection........................................................................................................48

6 RESULTS........................................................................................................................................50
   6.1 Test Setup................................................................................................................................50
   6.2 Network Monitoring Equipment.................................................................................................51
   6.3 Network Monitoring Output.........................................................................................................52

7 FUTURE WORK...............................................................................................................................57

8 SUMMARY.......................................................................................................................................58

REFERENCES.......................................................................................................................................59
List of Figures

Figure 1.1: Illustration of a Wireless Sensor Network (WSN)...............................................................11

Figure 1.2: Overview of a Sensor Node’s hardware components..........................................................12

Figure 4.1: Illustration of bifurcation of Sensor Node’s Firmware into Application and Bootloader in Flash memory.......................................................................................................................18

Figure 4.2: OTA Updater control flow..................................................................................................20

Figure 4.3: OTA Update Receiver control flow....................................................................................23

Figure 4.4: Illustration of the concept of Interrupt Vector Redirection [4]...........................................24

Figure 5.1: Nanett Evaluation Board....................................................................................................25

Figure 5.2: MSP-FET430UIF USB Debugging Interface.......................................................................26

Figure 5.3: State machine for OTA Updater type Bootloader.............................................................28

Figure 5.4: State machine for OTA Update Receiver type Bootloader..............................................31

Figure 5.5: OTA Network Packet data structure description.................................................................35

Figure 5.6: Description of Generic Packet Header...............................................................................36

Figure 5.7: Description of OTA_CONFIG data structure.................................................................37

Figure 5.8: Description of OTA_DATA data structure......................................................................38

Figure 5.9: Initial stages of the Application Upgradation Process.....................................................40

Figure 5.10: Data transfer stages of the Application Upgradation Process through OTA programming...41

Figure 5.11: Final stages of the Application Upgradation Process through OTA programming............42
Figure 5.12: Configuration Parameter List 1 for the OTA Updater type Bootloader..........................43

Figure 5.13: Configuration Parameters List 2 for the OTA Updater type Bootloader..........................44

Figure 5.14: Symbol Browser Window of CrossStudio IDE [6].................................................................45

Figure 5.15: Example of a Section Placement File.....................................................................................46

Figure 5.16: Memory Organization of MSP430F1611 microcontroller [8].................................................47

Figure 5.17: Bootloader Code Section Placement File..................................................................................47

Figure 5.18: Application Code Section Placement File.................................................................................48

Figure 6.1: Test setup of the Sensor Nodes...............................................................................................50

Figure 6.2: CC2531 USB Dongle [7]........................................................................................................51

Figure 6.3: SmartRF Packet Sniffer output of the initial stages of the Application Upgradation

   Process..................................................................................................................................................53

Figure 6.4: SmartRF Packet Sniffer output of the data transfer stages of Application Upgradation

   Process..................................................................................................................................................54

Figure 6.5: SmartRF Packet Sniffer output of the final stages of the Application Upgradation

   Process..................................................................................................................................................55
List of Tables

Table 3.1: Overview of various reprogramming protocols [2].................................................................16

Table 5.1: OTA network packet names and their corresponding types and defined values.......................35

Table 5.2: OTA Configuration packet types and defined values..........................................................39

Table 6.1: OTA network packet types, configuration types and their corresponding defined values.......52
Acronyms and Abbreviations

WSN  Wireless Sensor Network
OTA  Over The Air
IDE  Integrated Development Environment
ISR  Interrupt Service Routine
PAN  Personal Area Network
OMA  Open Mobile Alliance
XNP  Crossbow Network Programming
ISP  In-System Programming
MOAP  Multi-hop Over the Air Programming
MNP  Multi-hop Network reprogramming Protocol
CPU  Central Processing Unit
MCU  Micro Controller Unit
USB  Universal Serial Bus
JTAG  Joint Test Action Group
ANSI  American National Standards Institute
1. INTRODUCTION

1.1 Overview

The most familiar type of information processing has happened on substantial, general purpose computational devices, ranging from old-fashioned mainframes to modern compact mobile devices such as laptops and smartphones. In a lot of applications, for example, office applications, these computational devices are mostly employed to process information that is at its nucleus centered around a human user of a system, but is at best indirectly associated to the physical environment.

There exists another class of applications, where the physical environment is at the center of attention. Here, computation is utilized to apply control over physical processes, for example, when controlling the speed of a conveyer belt of a process chain in a production factory. In this scenario, the computation is integrated with the control, that is, it is embedded into a physical system. In contrast to the former class of systems, these embedded systems are usually not based on human interaction but are rather required to work independently. These systems are closely coupled to their control task in the context of a larger system.

These type of embedded systems are a well-known and long-used concept in the engineering sciences. Their influence on everyday life is also continuing to grow at a fast pace. Advancement in technology is paving way to take this spreading of embedded control in our daily lives a step further. In the near future, computation will encompass us in our daily lives, realizing a vision of “Ambient Intelligence” where many different devices will gather and process information from many different sources to both control physical processes and to interact with human users.
In order to accomplish this vision, a decisive aspect is needed in addition to computation and control. And this crucial link in the chain is none other than the communication. Each of these sources of information have to be capable to transfer the information to the place where it is needed, may it be an actuator or a user, and they should work jointly to provide as precise a picture of the real world as is required. In quite a few of the application scenarios, such networks of sensors and actuators are easily built using existing, wired networking technologies. But in many other application cases, however, the need to wire together all these entities creates a considerable obstacle to success. This is because wiring is expensive, given to the fact that a lot of interconnections is required. And this also adds up to the maintenance cost and effort. Hence, in many application scenarios, wireless communication between such devices is the best solution to have.

![Figure 1.1: Illustration of a Wireless Sensor Network (WSN)](image)

Consequently, a new class of networks has appeared in the past few years which is none other than the so-called Wireless Sensor Network (WSN). As illustrated in Figure 1.1, these networks consist of individual nodes that are able to interact with their environment by sensing or controlling physical parameters; these nodes have to collaborate to fulfill their tasks as, usually, a single node is incapable of doing so; and they use wireless communication to enable this collaboration [1]. Basically, the nodes
Without such a network contain functionalities such as computation, wireless communication, and sensing or control. Regardless of the fact that these networks also often include actuators, the term wireless sensor network has become the commonly accepted name. Quiet often, other names like “wireless sensor and actuator networks” are also found.

Typically, a sensor node comprises five main components as depicted in Figure 1.2.

**Controller:** A controller to process all the pertinent data and capable of executing arbitrary code.

**Memory:** Some type of persistent memory to store programs and intermediate data.

**Sensors and Actuators:** These are the actual interface to the environment where the sensor nodes are deployed. A means to observe or control physical parameters of the environment.

**Communication:** For nodes to form a network requires a device for sending and receiving information over a wireless channel.

![Figure 1.2: Overview of a Sensor Node’s hardware components](image)

WSNs such as these are potent in that they are flexible enough to support a lot of very different real-world applications. Due to this amenability, they are also a challenging research and engineering problem. In consequence, there is no single set of requirements that clearly classifies all WSNs, and there is also not a individual technical solution that incorporates the entire design space. For instance, in many WSN applications, individual nodes in the network cannot easily be connected to a wired power supply but
rather have to depend on onboard batteries. In these types of applications, the energy efficiency of any proposed solution is hence a very paramount figure of merit as a prolonged operation time is usually beneficial. In many other applications, power supply might not be a major concern and therefore other metrics such as the accuracy of the delivered results can become more significant. Additionally, the acceptable size and costs of an individual node can be apt in many applications.

Nowadays, in many of the applications, sensor networks are deployed once and are intended to operate unattended for a very long period of time. Management and maintenance tasks of WSNs are challenging due to the efforts involved. Empowering the sensor networks with the capability to be reprogrammed is a way to address such challenges. Reprogramming every such node in the network manually by individually plugging each one to a laptop or other mobile devices is not a feasible option. Furthermore, harsh environmental conditions often make the retrieval of the nodes for reprogramming nearly impossible. Traditional ways of manually reprogramming sensors are costly, labor intensive or even impossible since each node has to be collected from the field and physically attached to a computer to “burn” new codes. The protocols that have been developed for reprogramming wireless sensor networks are all specific to their individual platforms due to the differences in their architectures. Consequently, to make one of these protocols work on a different hardware platform, it often requires numerous software modifications. Therefore, reprogramming over the air (OTA) is imperative for many applications.

1.2 Report Organization

The following sections of the report is laid out as explained subsequently. The report starts off with the "State of the art" section which gives brief information about the current state of affairs in the field of OTA. Then the objectives and the goals of the thesis work is enumerated in the "Motivation and Objective" section and the difference between the existing solutions and the proposed solution in this thesis work is discussed. The "Concept" section explain the proposed concept in detail. Later on as proof of concept, the work related to the implementation of the concept on real hardware is described with supporting results in the "Implementation" section. Finally, the report ends with the possible future work on the concept and its implementation, and a comprehensive summary.
2. Motivation and Objective

Over-the-Air programming enables the software of sensor nodes to be updated via the transmission capability of the wireless sensor node itself instead of the traditional wired interface. As discussed in the previous section, all the developments that have been done and which are on-going are platform and operating system dependent. And there exists no standard for the development of OTA programming on resource constrained nodes where it is impractical to even have a small operating system running on it. All these mentioned reasons acted as an impetus for the development of a concept for OTA programming designed specifically for resource constrained nodes.

The proposed concept aims to achieve the following design objectives.

- Development of a concept for transmission and distribution of the firmware wirelessly
- Accomplishing the above mentioned task with optimized resource utilization
3. State of the Art

In the field of mobile phones communication, OTA is used to update a phone's operating firmware and this is very well established nowadays. Several standardization bodies were setup to help, develop, oversee, and manage OTA programming. One of them is the open mobile alliance (OMA) [14]. Hence currently, the concept of OTA in the domain of mobile phone communication has be well established.

A Variety of OTA protocols have been proposed for WSN in the past few years as discussed in [2]. The initial breakthrough on the contrary to the traditional way of reprogramming microcontrollers using In-System Programming (ISP) sensor networks that explored a way to update the node’s firmware without human intervention was developed at Berkeley, the University of California. Famously known as XNP (Crossbow Network Programming) is a one-hop protocol that makes available firmware updates through a wireless link [15]. Various protocols have been developed to achieve the goal of OAP including Deluge, MOAP and MNP. And some of the examples of the hardware platforms and operating systems these protocols are developed on include Mica, Telos and Imote, and TinyOS, Contiki and SOS, respectively. Due to the differences in their architectures, the protocols that have been developed are all platform specific.

Several reprogramming protocols have been designed and studied in the past few years. The summary of various protocols and their characteristics is presented in Table 3.1. Even though existing OAP protocols have many merits, they suffer from fundamental limitations that can critically impair their use in future systems. Primarily, the performance of existing OAP protocols quickly degrades as the network size and density increase, and even more so when packet loss is high. Unlike other protocols, these have to be designed very carefully. Moreover, the WSN has its own design and resource constraints. Resource constraints encompass a limited amount of energy, short communication range, low bandwidth and limited processing and storage space in each sensor node.
Since WSN is a multipurpose platform for many applications like structural health monitoring, security surveillance and pervasive computing, vigorous research is being carried out in this domain and we often come across many new protocols. The necessity of specific application forms the basis for protocol development in most of the cases.

<table>
<thead>
<tr>
<th>Name</th>
<th>Encoding/decoding</th>
<th>MAC</th>
<th>Hop</th>
<th>Scope</th>
<th>Hierarch</th>
<th>Pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td>XNP</td>
<td>Complete program</td>
<td>CSMA</td>
<td>Single hop</td>
<td>Whole network</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Rajen</td>
<td>Platform-dependent patch</td>
<td>CSMA</td>
<td>Single hop</td>
<td>Whole network</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Incremental</td>
<td>Platform-independent patch</td>
<td>CSMA</td>
<td>Single hop</td>
<td>Whole network</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Tiled</td>
<td>Male script</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>MOAP</td>
<td>Complete program</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Deluge</td>
<td>Complete program</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>MHP</td>
<td>Complete program</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Speekler</td>
<td>Complete program</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Firecracker</td>
<td>Complete program</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Infus</td>
<td>Complete program</td>
<td>TDMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Aqueduct</td>
<td>Complete program</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Selected nodes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Tnicoibus</td>
<td>Monitor update</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Selected nodes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Transaction-based TCL script</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>no</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>Raloois</td>
<td>Complete program</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Stream</td>
<td>Complete program</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Prestet</td>
<td>Complete program</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Synapse</td>
<td>Complete program</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Flash Memory</td>
<td>Monitor update</td>
<td>CSMA</td>
<td>Single hop</td>
<td>Whole network</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Commissioning</td>
<td>Monitor update</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Small World Concept</td>
<td>Complete program</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Stream ASC-RE*</td>
<td>Complete program</td>
<td>CSMA</td>
<td>Multi-hop</td>
<td>Whole network</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

*AS3, Application Support; RS, Reprogramming Support

Table 3.1: Overview of various reprogramming protocols [2]
4. Concept Description

Code dissemination and code acquisition are two basic schemes to reprogram sensor networks [3]. System administrators usually use code dissemination for updating programs on sensor nodes, fixing bugs, changing network functionality, tuning module parameters, and replacing program modules. In comparison to code dissemination, code acquisition originates from individual sensors to fetch and install program modules from the network dynamically and on demand. It enables sensor nodes to self-reprogram so that they can adapt to evolving environments and changing tasks accordingly. Due to memory constraints of sensor nodes, a monolithic program with too many functions cannot be incorporated into the memory. In addition, applications may need extra memory for modules to incorporate handling capabilities for unforeseen events. In this work we are going to be using the code dissemination scheme to reprogram the sensor nodes. Also, the code dissemination is employed only with a single hop in the network. That is, we can only do code dissemination for a single node at a time.

4.1 Firmware Bifurcation

In order to give the sensor node the capability to be able to be reprogrammed, we are making use of the concept of Bootloader. This approach leads to the bifurcation of the firmware that will be going into the sensor node. That is, the firmware is divided into two parts, namely, the "Bootloader" and "Application". Figure 4.1 illustrates the design clearly.

A Bootloader is the code that will be loaded into memory and becomes the main application code running on the microcontroller during normal operation. It resides on the microcontroller forever in a reserved portion of on-board Flash memory. The Bootloader can be updated only by connecting the device directly to a host system by a wired connection and by reprogramming it manually. It is this on-board Bootloader that makes it possible for a sensor node's application code to be updated in the field. The Bootloader does not have any code related to the functionality related to the application of the sensor
node. The vector table and the reset vector also reside close to the sectors having the Bootloader. Only the Bootloader has the capability to reprogram the Application code other than the manual reprogramming from the host machine. The Bootloader also ensures that the flash memory sectors containing the Bootloader, Vector Table and the Reset Vector are not tampered when the Application code is being changed. This makes sure that whenever the device is reset, it always runs the Bootloader even if the Application code is corrupt.

Figure 4.1: Illustration of bifurcation of Sensor Node’s Firmware into Application and Bootloader in Flash memory.
The Application code is the one which encompasses the actual code dealing with the functionality related to the application of the sensor node. This code is deliberately made to reside in a separate portion of the flash memory other than the Bootloader. It is made sure that this code is placed in different memory sectors so that when a flash erase is performed, then the sectors containing the Bootloader code is not affected by this.

In this concept, we take the approach where upgrading the Application code in one sensor node is performed by another sensor node. This has the advantage of eliminating the need of having a host machine to establish a wireless connection with a sensor node in order to update its Application code. In order to make this approach feasible, we are defining two types of Bootloader, namely, the "OTA Updater" and "OTA Update Receiver". The new Application code that needs to be transmitted wirelessly to other sensor nodes is first loaded into a sensor node which also contains the OTA Updater type Bootloader. The OTA Updater type Bootloader has the responsibility of transmitting this new Application code to the destination sensor node. After a successful transfer of the Application code to the sensor nodes containing the OTA Update Receiver type bootloader, the new Application starts to execute.

4.2 OTA Updater type Bootloader

The sensor node containing the OTA Updater type Bootloader is the node which is going to be initiating an Application update in another sensor node. Hence, along with the OTA Updater type Bootloader code, this sensor node is also loaded with the new Application code that will be used to update the one in the other sensor nodes. Also, all required information related to the Application code placement in the destination sensor node is made available to the OTA Updater type Bootloader. One of the fields in this information is the address of the "main" function of the Application code from where the execution of the code begins. This is very important because once the Application code is updated, a code jump is performed from the Bootloader code to the Application code, after which the Application code starts executing.
Figure 4.2: OTA Updater control flow
The flow chart in the Figure 4.2 illustrates the control flow of the OTA Updater type Bootloader. When the user signals the OTA Updater to initiate the OTA update, the Bootloader enters the OTA updater mode. Once inside this mode, the OTA Updater type Bootloader tries to communicate with the OTA Update Receiver type Bootloader in order to initiate the update procedure at the other end. After relevant informations are exchanged between them, the actual update procedure begins. At the end, after the completion of the update, the OTA Updater sends a message to the OTA Update Receiver indicating the end of update and comes out of the OTA updater mode.

4.3 OTA Update Receiver type Bootloader

The sensor node containing the OTA Update Receiver type Bootloader is the node which is going to be experiencing the updating of its Application code. This behavior of the node is triggered when an appropriate message is received from the OTA Updater. Until then this sensor node will be executing its currently residing Application code. Once all the relevant information is obtained from the OTA Updater, the OTA Update Receiver node will start updating the Application code. And after this process, a switch to the new Application code is made by performing a code jump to the start address of the "main" function of the new Application code.

The flow chart in the Figure 4.3 illustrates the control flow of the OTA Update Receiver type Bootloader. Whenever the device receives an OTA update request from OTA Updater, it stores the request in some permanent memory and resets itself to enter the Bootloader. Here in the OTA Update Receiver type Bootloader mode, first a check is performed to see whether a request has been received previously for initiating update procedure. If that’s the case, then the Bootloader enters the OTA Update Receiver mode. Or else, the bootloader jumps to the already present Application code. Once inside the OTA Update Receiver mode, the bootlader waits until it has received all the relevant information to start the update procedure. Only then the actual update is initiated. After the completion of the update, a message indicating the completion of the update procedure is sent to the OTA Updater. Later on the bootloader jumps to the newly updated Application code.
Figure 4.3: OTA Update Receiver control flow
4.4 Communication

The communication between the sensor nodes is established by means of the radio transceivers which are present in the sensor nodes. A distinctive class of packet structure is created to facilitate the Application update procedures in the sensor nodes. The network packets with this packet structure is utilized to send and receive control and data packets between the sensor nodes which are programmed to act as the OTA Updater and the OTA Update Receiver.

4.5 Effects of Firmware Bifurcation

One of the major challenges arises when we bifurcate the firmware as Bootloader and the Application. That is none other than the utilization of the interrupt service routine once we make a switch from Bootloader code execution to the Application code execution. To allow interrupts on an application level, a software vector redirection method can be implemented. To achieve this we fix the contents of the default vector table and point to a Proxy table that resides in application space. Figure 4.4 illustrates this concept clearly.

The concept of Interrupt Vector Redirection can be understood with the help of the following steps [4].

1. The application receives an interrupt request, the current address is pushed into the stack, and the CPU fetches the address from the Vector Table.
2. The Vector Table contains the address of the proxy location for each interrupt. The CPU fetches the address of the corresponding entry in the Proxy Table and jumps to it.
3. The Proxy Table contains branch instructions (BRA) followed by the actual address of each ISR. The CPU executes the BRA instruction and jumps to the corresponding application vector ISR.
4. Upon completion of the ISR, the RETI instruction is executed, and the previous address is popped from the stack.

This process is almost transparent for the implementation of an application, but it is important to note that there is added latency due to the additional jump from the Proxy Table to the application ISR.
Most of the microcontrollers support vector redirection at the hardware level which makes the above procedure to be implemented very easily. Whereas in some microcontrollers this support is not available and hence it is difficult to implement the above procedure. This is also because of the fact that it requires extensive support from the integrated development environment to have a finer control over the software development for the microcontroller to accomplish the above procedure.
In order to validate the proposed concept for the Over The Air programming of the sensor nodes the following set of hardware and software components were employed. For the hardware aspect, the ultra-low-power sensor network evaluation platform of the “nanett” project (Nano System Integration Network of Excellence) was chosen and for the development of the software the “CrossStudio with CrossWorks for MSP430” [10] Integrated Development was utilized. The software development is carried out in C programming language.

5.1 Hardware

5.1.1 Nanett Evaluation Board

![Nanett Evaluation Board](image-url)
The Nanett Evaluation Board is the ultra-low-power sensor network evaluation platform of the nanett project (Nano System Integration Network of Excellence) developed at Technische Universität Chemnitz. At the heart of this evaluation board there lies a microcontroller, namely, MSP430F1611, which belongs to the Texas Instruments MSP430 family of ultralow power microcontrollers [5]. It has the capability to run at 8MHz and 32.768kHz for high speed and low speed modes, respectively. It has an internal flash memory of 48KB and a RAM of 10KB, which can be used both as data and code memory. The evaluation board is equipped with a IEEE 802.15.4 standards compliant radio transceiver, CC2520, from Texas Instruments. The power supply to the board can be either a 3.6V Li-polymer battery or a direct 3V dc supply. The board is designed such that the wake-up and receive power consumption is less than 100 micro amps.

5.1.2 MSP-FET430UIF USB Debugging Interface

![MSP-FET430UIF USB Debugging Interface](image-url)
The MSP-FET430UIF is a powerful flash emulation tool to quickly begin application development on the MSP430 MCU [11]. It includes USB debugging interface used to program and debug the MSP430 in-system through the JTAG interface or the pin saving Spy Bi-Wire (2-wire JTAG) protocol. The flash memory can be erased and programmed in seconds with only a few keystrokes, and since the MSP430 flash is ultra-low power, no external power supply is required. The debugging tool interfaces the MSP430 to the CrossStudio integrated development environment. This device is used to burn the developed software into the microcontroller memory, may it be the Bootloader or the new Application which needs to be transferred wirelessly from the Bootloader.

5.2 Software

5.2.1 CrossStudio with CrossWorks for MSP430

CrossWorks for MSP430 is a complete C development system for all the Texas Instruments MSP430 based microcontrollers. The toolset provides a complete and cost-effective solution for programming the MSP430 family of low-power microcontrollers. It is bundled with an ANSI C compiler, macro assembler, linker/locator, libraries, core simulator, flash downloader, JTAG debugger, and an integrated development environment, CrossStudio. The version of the software used is “CrossWorks for MSP430 Version 2.2.1”. The software used for the concept implementation is entirely written in ANSI C programming language.

5.3 Implementation of Bootloader State Machines

5.3.1 OTA Updater type Bootloader State Machine
Figure 5.3: State machine for OTA Updater type Bootloader

**INITIATE OTA**

This is the state in which the state machine of the OTA Update type Bootloader will be initialized with. This state is entered only when the “Button 1” of the sensor node is activated by the user. Once this state is entered, the OTA Updater type Bootloader starts sending the “INITIATE OTA” type network packet to the destination sensor node, whose address is made available beforehand during the build process of
the code for the OTA Updater type Bootloader. After successfully sending this request network packet, the OTA Updater type Bootloader waits for the sensor node with OTA Update Receiver type Bootloader to respond with an acknowledgement network packet. The sending of this request network packet is continued up until a response is received from the OTA Update Receiver type Bootloader. As soon as this acknowledgement network packet is received from the sensor node with OTA Update Receiver type Bootloader, the state change from “INITIATE_OTA” to “CONFIG_OTA” is performed.

**CONFIG_OTA**

Once this state is entered, the OTA Update type Bootloader starts sending “CONFIG_OTA” type network packet to the destination sensor node. The “CONFIG_OTA” type network packet contains all the relevant information required by the OTA Update Receiver type Bootloader to start the upgradation process of its Application code. This information includes the memory size and the address of the Code Section which is going to be transmitted in the later state. Again the sending of this network packet is continued up until a response is received from the OTA Update Receiver type Bootloader. As soon as this acknowledgement network packet is received from the sensor node with OTA Update Receiver type Bootloader, the state change from “CONFIG_OTA” to “SEND_UPGRADE_DATA” is performed. The “CONFIG_OTA” state is revisited whenever a change in Code Section, which is to be transmitted, is detected. There can exits more than one Code Section for a particular Application Code which differ in their memory size and addresses. And hence, in order to update this new configuration at the OTA Update Receiver type Bootloader, this state is revisited as much time as it is required.

**SEND_UPGRADE_DATA**

The “SEND_UPGRADE_DATA” is the state where the actual data related to the upgradation of the Application Code is transmitted sequentially. During the sending of the data related to each Code Section of an Application Code, the information related to the remaining Code Sections to be transmitted is monitored regularly. If the sending of a particular Code Section is finished, then a check is performed to see if there exits any more Code Sections that need to be transmitted. If this check turns out to be true,
then a state change is performed from “SEND_UPGRADE_DATA” to “CONFIG_OTA”. This is necessary to convey the updated configuration information of the next Code Section to be transmitted to the OTA Update Receiver type Bootloader. After all the Code Sections of the Application Code have been transmitted successfully, a state change is performed from “SEND_UPGRADE_DATA” to “DATA_SEND_FINISH_OTA”.

**DATA_SEND_FINISH_OTA**

This is the final state that the OTA Update type Bootloader will be entering during the Application Code upgradation process. In this state, the OTA Update type Bootloader just sends the “DATA_SEND_FINISH_OTA” type network packet to the sensor node with OTA Update Receiver type Bootloader. One of the information that is transmitted in this network packet type is the start address of the “main” function of the Application Code. This is an important information that is required by the OTA Update Receiver type Bootloader to configure the jump location in order to switch the control flow from Bootloader to the Application Code. Hence, this network packet indicates the OTA Update Receiver type Bootloader that all the data pertaining to the Application Code upgradation has been sent successfully. After sending this network packet, the OTA Update type Bootloader waits for the “Button 1” to be activated, which will lead to the start of the Application Code upgradation process once again.

**5.3.2 OTA Update Receiver type Bootloader State Machine**
Figure 5.4: State machine for OTA Update Receiver type Bootloader
APPLICATION_MODE

The “APPLICATION_MODE” is the state that the sensor node will normally be in. Normal functionality of the sensor node which is defined in the Application Code will be under execution in this state. The network packet with type “INITIATE_OTA” conveys the message that a request to initiate the Application Code upgradation process has been put forward. Hence, whenever an “INITIATE_OTA” type network packet, which is addressed to this particular sensor node, is received, the sensor node saves this request in permanent memory and executes a code to software reset the device. This software reset is performed to enter the Bootloader Mode of the sensor node.

ACK_FOR_INITIATE_OTA

Whenever the sensor node is started up afresh, in accordance to the design, the OTA Update Receiver type Bootloader will be the code that will be running first. Here, in the “ACK_FOR_INITIATE_OTA” state, a check is performed to ascertain whether a request to initiate the Application Code upgradation process has been received previously or not. If that request is not received, then the code flow is transferred to the Application Code by first retrieving the saved information of its jump location from the permanent memory. This request is registered whenever an “INITIATE_OTA” type network packet is received from the sensor node having the OTA Updater type Bootloader. If the request to initiate the Application Code upgradation process has been validated, then an acknowledgement network packet of type “ACK_FOR_INITIATE_OTA” is sent to the sensor node having the OTA Updater type Bootloader, indicating that this sensor node is ready to service the request. And a state change from “ACK_FOR_INITIATE_OTA” to “WAIT_FOR_CONFIG_OTA” is performed.

WAIT_FOR_CONFIG_OTA

After acknowledging the request for the Application Code upgradation process in the “ACK_FOR_INITIATE_OTA” state, the sensor node ends up waiting in this state for the sensor node with the OTA Updater type Bootloader to send the relevant configuration information to begin the upgradation process. Once this configuration information is received, appropriate preparations are performed to start storing the data that will be received in the future packets. This preparation includes deciding whether the data that will be received in the future packets are destined to be written into Flash memory or the
RAM. If the destination is Flash memory, then the Flash memory needs to be prepared for the write operation by erasing the appropriate sectors. If the destination is RAM, then no additional preparation is required for writing into it. If the destination location is found to be inappropriate, then a state change is performed from “WAIT_FOR_CONFIG_OTA” to “ERROR_OTA”. Else, after all the preparation is finished successfully, an acknowledgement packet of type “ACK_FOR_CONFIG_OTA” is sent to the sensor node which had previously sent the configuration data. Then a state change is performed from “WAIT_FOR_CONFIG_OTA” to “ACK_FOR_DATA_OTA”.

ACK_FOR_DATA_OTA

In this state, first the data packet received is analyzed. Then appropriate decision is taken to write the data into Flash memory or RAM based on the analysis. The received data packets contain the relevant information regarding the destination where the data has to be written into and also the size of the data elements sent in the packet. This helps in the previously mentioned analysis. During the midst of the reception of these data packets, if a packet of type “CONFIG_OTA” is received, then this indicates that the transmission of one Code Section of the Application Code is finished and another Code Section configuration details has been sent. Hence, a state change from “ACK_FOR_DATA_OTA” to “WAIT_FOR_CONFIG_OTA” is performed. This is required in order to carry out appropriate preparations to start storing the next Code Section’s data that will be received in the future packets. Once all the Code Sections of the Application Code have been transferred by the sensor node with OTA Updater type Bootloader, it transmits a network packet of type “DATA_SEND_FINISH_OTA”. As soon as this type of network packet is received by the sensor node with the OTA Update Receiver type Bootloader, a state change is performed from “ACK_FOR_DATA_OTA” to “ACK_FOR_DATA_SEND_FINISH_OTA”.

ACK_FOR_DATA_SEND_FINISH_OTA

After arriving in this state the OTA Update Receiver type Bootloader copies the start address of the “main” function of the Application Code, which it had received from the “DATA_SEND_FINISH_OTA” type network packet earlier, into the permanent memory. Then the network packet of type “ACK_FOR_DATA_SEND_FINISH_OTA” is sent to sensor node with the OTA Updater type Bootloader to indicate that the upgrade process is completed. Finally, a state change is performed from
“ACK_FOR_DATA_SEND_FINISH_OTA” to “ACK_FOR_INITIATE_OTA”, thereby going to the first state of the OTA Update Receiver type Bootloader.

ERROR_OTA

This is a state to indicate error conditions. This state is reached whenever the validation of the configuration data is evaluated to be negative in the “WAIT_FOR_CONFIG_OTA” state. Hence, a network packet of type “ERROR_OTA” is sent to the sensor node with OTA Updater type Bootloader to indicate the error condition that has occurred in the Application Upgrade process. After intimating the sensor node with OTA Updater type Bootloader, a state change is performed from “ERROR_OTA” to “ACK_FOR_INITIATE_OTA”. Thereby, waiting for the new Application Upgrade process to be re-initiated.

5.3.3 Bootloader Memory Requirement

The Bootloader code development was carried out in accordance with the objectives of this Thesis work that is to keep the memory footprint as low as possible. The overall memory requirement of the developed Bootloader comes up to 6.5 kilo bytes, which includes 6 kilo bytes of Code memory and 0.5 kilo bytes of Data memory. This assures that there is enough memory left for the usage by the Application Code.

5.4 OTA Network Packet Types and Structure

A handful of network packets are used to carry out the Application upgradation process. And these are vital for maintaining the communication between the sensor nodes with OTA Updater and OTA Update Receiver type Bootloaders. The Table 5.1 enumerates the types of OTA network packets used and classifies them to their corresponding types with the assigned numerical constant values.
Table 5.1: OTA network packet names and their corresponding types and defined values

<table>
<thead>
<tr>
<th>OTA Network Packet Name</th>
<th>OTA Network Packet Type</th>
<th>Values Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTA_PACKET_TYPE_CONFIG</td>
<td>Configuration</td>
<td>0x11</td>
</tr>
<tr>
<td>OTA_PACKET_TYPE_DATA</td>
<td>Data</td>
<td>0x13</td>
</tr>
</tbody>
</table>

```c
typedef struct _OTA_PACKET_STRUCT {
  union {
    unsigned char data[108];
    struct {
      //--------HEADER----------
      GENERIC_PACKET_HEADER
      //--------PAYLOAD---------
      unsigned char ota_packet_type; // Indicates Config / Data
    }
    type_u;
  }
  data_struct;
} data_u;
OTA_PACKET;
```

Figure 5.5: OTA Network Packet data structure description
The Figure 5.5 illustrates the contents of the OTA network packet structure that is utilized in the implementation of this concept. This is the type of packet that is used to communicate both configuration and data information between the sensor nodes having OTA Updater type Bootloader and OTA Update Receiver type Bootloader. The packet structure starts with the Generic Packet header as described in the Figure 5.6, which can be used for routing purposes. This field does not hold any physical significance in the present implementation. But it is include so that it could be used in the future developments. This entire packet structure is embedded in the payload section of the IEEE 802.15.4 standard which is the protocol used for network communication here. Hence, the payload section of the IEEE 802.15.4 network packet will be 108 bytes of length in accordance to the OTA Network Packet data structure.

The next field in the structure is “ota_packet_type” which is of type “unsigned character”. This field indicates whether the OTA packet is of Configuration Type or Data Type. The question of field comes next in the OTA packet structure is dependent on the “ota_packet_type” field.

```c
#define GENERIC_PACKET_HEADER __attribute__((packed))
    
    unsigned char packetType; \
    unsigned char subType; \n    unsigned char ttl; \
    unsigned char payloadsize;\n    unsigned short srcId; \n    unsigned short dstId;
```

Figure 5.6: Description of Generic Packet Header
typedef struct _OTA_CONFIG_STRUCT{
    unsigned char ota_update_availability;
    unsigned char ota_session_id;
    unsigned char data_write_location_upper_byte;
    unsigned char data_write_location_lower_byte;
    unsigned char total_data_size_upper_byte;
    unsigned char total_data_size_lower_byte;
    unsigned char ack_to_sequence_nr;
} OTA_CONFIG;

Figure 5.7: Description of OTA_CONFIG data structure

In the OTA network packet structure when the “ota_packet_type” field is of type Configuration, the next field is defined as “OTA_CONFIG”, which is a structure that encompasses all the configuration information required for the Application upgradation process. This is clearly illustrated in the Figure 5.7.

The significance of each of the fields of this structure is explained as follows.

ota_update_availability : Indicates whether the OTA update is available or not.

ota_session_id : Indicates the Session Identification Number which is unique for every Application upgradation session. This is used to keep the communication only between the parties involved.

data_write_location_upper_byte & data_write_location_lower_byte : Indicates the address of the memory location where the Code Data that will be sent is intended to be written into.

total_data_size_upper_byte & total_data_size_lower_byte : Indicates the total size of the Code Section in bytes that is going to be sent after splitting it up to fit it into the data packets.

ack_to_sequence_nr : Indicates the acknowledgement that is being sent to the previous packet received. It just holds the sequence number of the previous packet received.
typedef struct _OTA_DATA_STRUCT{
    unsigned char ota_session_id;
    unsigned char data_size_sent_in_packet;
    unsigned char data[97];
} OTA_DATA;

Figure 5.8: Description of OTA_DATA data structure

In the OTA network packet structure when the “ota_packet_type” field is of type Data, the next
field is defined as “OTA_DATA”, which is a structure that encompasses the upgrade data which will be
written into the memory. This is clearly illustrated in the Figure 5.8. The significance of each of the fields
of this structure is explained as follows.

ota_session_id : Indicates the Session Identification Number which is unique for every Application
upgradation session. This is used to keep the communication only between the parties involved.

data_size_sent_in_packet : Indicates the size of data in bytes that is being sent in the present packet.

data[97] : This is the array of data bytes that is being transmitted for Application upgradation. A maximum
of 97 bytes can be transmitted in each data packet.

Table 5.2 enumerates all the different types of Configuration packets used and also lists out the
designated numerical constant values for them.
Table 5.2: OTA Configuration packet types and defined values

<table>
<thead>
<tr>
<th>OTA Configuration Packet Types</th>
<th>Values Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTA_UPDATE_AVAILABLE</td>
<td>0x21</td>
</tr>
<tr>
<td>OTA_UPDATE_NOT_AVAILABLE</td>
<td>0x23</td>
</tr>
<tr>
<td>OTA_UPDATE_READY_FOR_UPDATE</td>
<td>0x25</td>
</tr>
<tr>
<td>OTA_UPDATE_START</td>
<td>0x27</td>
</tr>
<tr>
<td>OTA_UPDATE_IN_PROGRESS</td>
<td>0x29</td>
</tr>
<tr>
<td>OTA_UPDATE_END</td>
<td>0x31</td>
</tr>
<tr>
<td>OTA_UPDATE_ERROR</td>
<td>0x33</td>
</tr>
</tbody>
</table>

5.5 OTA Network Packet Exchange Progression

As a result of the interaction between the OTA Updater and OTA Update Receiver nodes during the Application Upgradation process, a distinctive network activity which includes exchange of packets between them can be observed. The following figures, namely, Figure 5.9, Figure 5.10 and Figure 5.11 depict the exchange of the OTA network packets from the start of the Application Upgradation process up until its termination.
The Figure 5.9 depicts the early stages of the Application Upgradation Process. Here we can observe the packet exchange from the initialization state and the state where the first configuration data is transmitted prior to actual data transfer. The acknowledge packets transmitted for each packet received is also clearly visible.

The Figure 5.10 depicts the mid stages of the Application Upgradation Process. Here we can observe the data packet transfers and the acknowledgements received for them as well. The figure also shows the configuration packet delivery and reception of acknowledgement for the same in between data packet transfers.
Figure 5.10: Data transfer stages of the Application Upgradation Process through OTA programming.
Figure 5.11: Final stages of the Application Upgradation Process through OTA programming.

The Figure 5.11 depicts the final stage of the Application Upgradation Process. Here we can observe the Application Upgradation process termination packet being sent and also the reception of the appropriate acknowledgement packet.

5.6 Configuration Parameters for OTA Updater type Bootloader

In order to take care of the correct placement of the new Application Code and to retrieve it from the right memory location, we need to provide some configuration parameters before the Bootloader Code is built for execution. These configurations are the one which help in avoiding the over writing of the memory sectors containing Bootloader, Vector Tables and Reset vectors.
Figure 5.12: Configuration Parameter List 1 for the OTA Updater type Bootloader

The Figure 5.12 and Figure 5.13 enumerates all of the configuration parameters required by the OTA Updater type Bootloader. The description of each of this configuration parameter is as follows.

**RAM_START_ADDR_AVAILABLE_FOR_APP** : Indicates the start address of the RAM location from which it is freely available for use by the Application Code. This address can be obtained from the CrossStudios IDE after the build process of the Bootloader Code.

**RAM_END_ADDR_AVAILABLE_FOR_APP** : Indicates the end address of the RAM location after which it is not available for use by the Application Code. This address can be obtained from the CrossStudios IDE after the build process of the Bootloader Code.

**FLASH_START_ADDR_AVAILABLE_FOR_APP** : Indicates the start address of the Flash memory location from which it is freely available for use by the Application Code. This address can be obtained from the CrossStudios IDE after the build process of the Bootloader Code.

**FLASH_END_ADDR_AVAILABLE_FOR_APP** : Indicates the end address of the Flash memory location after which it is not available for use by the Application Code. This address can be obtained from the CrossStudios IDE after the build process of the Bootloader Code.

**BOOTLOADER_START_ADDR** : Indicates the start address of the Bootloader. This address can be obtained from the CrossStudios IDE after the build process of the Bootloader Code.

**ADDR_OF_APPLICATION_MAIN_FUNCTION_UPPER_BYTE** & **ADDR_OF_APPLICATION_MAIN_FUNCTION_LOWER_BYTE** : Indicates start address of the “main” function of the Application Code. This address can be obtained from the the Symbol Browser Window of CrossStudio IDE (Figure 5.11) after the build process of the Application Code.
Figure 5.13: Configuration Parameters List 2 for the OTA Updater type Bootloader

**NUMBER_OF_SECTIONS_TO_SEND**: Indicates the number of Code Sections of the Application Code that are meant to be transmitted.

**IDATA0**: Indicates one of the Code Sections of the Application Code, namely, Initialized Data Section.

**UDATA0**: Indicates one of the Code Sections of the Application Code, namely, Uninitialized Data Section.
**CODE**: Indicates one of the Code Sections of the Application Code, namely, Code Section, where the actual instructions are located.

**CONST**: Indicates one of the Code Sections of the Application Code, namely, Constants Section, where the constants used for by Code Section is located.

The rest of the parameters, as their name suggests, indicate start addresses and the total size of that particular Code Section, respectively. These parameters can be obtained from the Symbol Browser Window of CrossStudio IDE (Figure 5.14) after the build process of the Application Code.

![Symbol Browser Window of CrossStudio IDE](image)

**Figure 5.14: Symbol Browser Window of CrossStudio IDE [6]**

Finally, the code development is done in such a way that the same source code is used as the base for both the OTA Updater and OTA Update Receiver type Bootloaders. But the relevant code segregation is done in the source code with the help of Preprocessor Directives. The following preprocessor directives need to be defined for the proper functioning of the Bootloaders in the files, namely, “ce-eeprom.h”. For the OTA Updater type Bootloader the preprocessor directive “#define OTA_UPDATER” has to be defined and for the OTA Update Receiver type Bootloader the preprocessor directive “#define OTA_UPDAT_RECEIVER” has to be defined.
5.7 Section Placement File

The concept discussed earlier in the report regarding bifurcation of the Firmware into the Bootloader and the Application Code can only be realized if we have precise control over the memory locations at which these two are placed during the programming of the microcontroller. This can be achieved with the help of Section Placements Files in the CrossStudio IDE.

```xml
<Root name="Flash Section Placement">
  <MemorySegment name="FLASH">
    <ProgramSection name=".vectors" load="Yes" />
    <ProgramSection name=".text" load="Yes" />
  </MemorySegment>
  <MemorySegment name="SRAM">
    <ProgramSection name=".stack" load="No" />
  </MemorySegment>
</Root>
```

Figure 5.15: Example of a Section Placement File

The Section Placement File is a XML-formatted file and it specifies where to place program sections in the target's memory segments. For example, a section-placement file that places a section called .stack in the SRAM segment and the .vectors and .text sections in the FLASH segment would look like the one depicted in Figure 5.15.
Figure 5.16: Memory Organization of MSP430F1611 microcontroller [8]

<!DOCTYPE Linker_Placement_File>
<Root name="MSP430 Section Placement">
  <MemorySegment name="RAM">
    <ProgramSection load="Yes" name=".abs" />
    <ProgramSection name="IDATA0" />
    <ProgramSection name="UDATA0" />
    <ProgramSection name="THREAD" />
  </MemorySegment>
  <MemorySegment name="INFO_A">
    <ProgramSection load="Yes" name="INFO_A" />
  </MemorySegment>
  <MemorySegment name="INFO_B">
    <ProgramSection load="Yes" name="INFO_B" />
  </MemorySegment>
  <MemorySegment name="FLASH">
    <ProgramSection load="Yes" size="6000" start="0xE000" name="CODE" />
    <ProgramSection load="Yes" size="528" start="0xF800" name="CONST" />
    <ProgramSection load="Yes" size="512" start="0xFB00" name="ISR" />
    <ProgramSection load="Yes" size="32" start="0xFFE0" name="INTVEC" />
  </MemorySegment>
</Root>

Figure 5.17: Bootloader Code Section Placement File
The Section Placement Files used for the build process of the Bootloader and the Application Code is depicted in the Figure 5.17 and Figure 5.18, respectively. These Section Placement files are modified according to the memory organisation of the MSP430F1611 microcontroller which is shown in Figure 5.16.

5.8 Interrupt Vector Redirection

The concept of Interrupt Vector Redirection could not be realized in this implementation because of two reasons. The first reason being that there is no hardware vector redirection support available in the microcontroller that is employed in our evaluation board. And secondly, in order to implement the vector redirection concept in the software, a comprehensive support is needed from the integrated development environment (IDE), which includes working closely with the compilation and linking phases. But the license acquired for the IDE that we have used here did not permit us to gain that insight into the
system. Hence, we adopted a much simpler approach of not using any Interrupt Service Routines (ISR) in the Application Code.

The driver code required for the successful operation of the radio transceiver resides in the Bootloader Code. Hence, all the ISRs used by this driver code is enabled only in the Bootloader Code. The application code utilized is a simple code which does not use any ISRs. This approach proved sufficient for the validation of the Application Upgrade Procedure by the OTA programming method that we have proposed in this report.
6. Results

6.1 Test Setup

The test setup used to validate the implementation of the proposed concept is depicted in the Figure 6.1. It consists of two sensor network evaluation platform of the nanett project [12]. One of the sensor node is programmed with the OTA Updater type Bootloader and the other one is programmed with the OTA Update Receiver type Bootloader. In addition to the Bootloader Code, the sensor node with OTA Updater type Bootloader is also programmed with the new Application Code that will be used for the upgrade procedure. The sensor node with the OTA Update Receiver type Bootloader also contains an
Application Code which will be erased once the upgrade procedure begins. Hereafter we will address the sensor node with the OTA Updater type Bootloader as “OTA Updater” and the sensor node with the OTA Update Receiver type Bootloader as “OTA Update Receiver”. The communication protocol utilized by the wireless transceivers is compliant to the IEEE 802.15.4 standard.

6.2 Network Monitoring Equipment

![Figure 6.2: CC2531 USB Dongle [7]](image)

In order to capture the network activity taking place in between the OTA Updater and the OTA Update Receiver after the Application upgrade process as begun, we make use of a wireless network packet sniffer. The CC2531 USB Dongle [7] depicted in the Figure 6.2 is from Texas Instruments. The CC2531USB-RD provides a PC interface to 802.15.4 / ZigBee applications. The dongle can be plugged directly into a PC and can be used as an IEEE 802.15.4 packet sniffer.
The SmartRF Packet Sniffer [13] is the required PC software application that can display and store radio packets captured by a listening RF device. The Packet Sniffer filters and decodes packets and displays them in a convenient way, with options for filtering and storage to a binary file format. The option to capture the IEEE 802.15.4/ZigBee packets is selected when we run the SmartRF Packet Sniffer software.

### 6.3 Network Monitoring Output

The figures 6.3, 6.4 and 6.5 show the split-up image of the network packets captured in the SmartRF Packet Sniffer software. It clearly shows all the packets exchanged between the OTA Updater and OTA Update Receiver during different stages of the Application Upgrade procedure. The Addresses of the OTA Updater and OTA Update Receiver are “0xE664” and “0x4C5C”, respectively, which can be observed in the Source and Destination Address fields in the output. The Destination PAN used is “0x1111”, which is and should be same for the packets to be exchanged between the sensor nodes. We can also observe the Sequence Number of each packet and the complete payload of the network packet which is 108 bytes in length, in accordance with the OTA Network Packet data structure. The Table 6.1 can be used to identify the packets being exchanged.

<table>
<thead>
<tr>
<th>OTA Network Packet Name</th>
<th>OTA Network Packet Type</th>
<th>Values Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTA_PACKET_TYPE_CONFIG</td>
<td>Configuration</td>
<td>0x11</td>
</tr>
<tr>
<td>OTA_PACKET_TYPE_DATA</td>
<td>Data</td>
<td>0x13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OTA Configuration Packet Type</th>
<th>OTA Configuration Packet Type</th>
<th>Values Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTA.UPDATE_AVAILABLE</td>
<td>Configuration</td>
<td>0x21</td>
</tr>
<tr>
<td>OTA.UPDATE_NOT_AVAILABLE</td>
<td>Configuration</td>
<td>0x23</td>
</tr>
<tr>
<td>OTA.UPDATE_READY_FOR_UPDATE</td>
<td>Configuration</td>
<td>0x25</td>
</tr>
<tr>
<td>OTA.UPDATE_START</td>
<td>Configuration</td>
<td>0x27</td>
</tr>
<tr>
<td>OTA.UPDATE_IN_PROGRESS</td>
<td>Configuration</td>
<td>0x29</td>
</tr>
<tr>
<td>OTA.UPDATE_END</td>
<td>Configuration</td>
<td>0x31</td>
</tr>
<tr>
<td>OTA.UPDATE_ERROR</td>
<td>Configuration</td>
<td>0x33</td>
</tr>
</tbody>
</table>

Table 6.1: OTA network packet types, configuration types and their corresponding defined values
Figure 6.3: SmartRF Packet Sniffer output of the initial stages of the Application Upgradation process
Figure 6.4: SmartRF Packet Sniffer output of the data transfer stages of Application Upgradation process
Figure 6.5: SmartRF Packet Sniffer output of the final stages of the Application Upgradation Process
Figure 6.3 shows the captured network packets in the SmartRF Packet Sniffer software which are related to the initial stages of the Application Upgradation process. The initial 2 packets are sent from OTA Updater indicating an Application Update is available. Once the OTA Update Receiver gets a hold of this type of message it replies back with an acknowledgement packet. Later after receiving the acknowledgement the OTA Updater sends the configuration packet. Again, the OTA Update Receiver replies back to this message with an appropriate acknowledgement packet. This finishes the initial setup phase and makes way for the data transfer in the future communications.

Figure 6.4 shows the captured network packets in the SmartRF Packet Sniffer software which are related to the data transfer stages of Application Upgradation process. We can clearly observe here that with each data packet received from the OTA Updater, the OTA Update Receiver replies back with an appropriate acknowledgement packet.

Figure 6.5 shows the captured network packets in the SmartRF Packet Sniffer software which are related to the final stages of the Application Upgradation Process. Here we can notice that just as soon as the data to be sent is finished, an intimation about the same is sent from the OTA Updater. The OTA Update Receiver acknowledges this by sending a reply packet. Then finally the OTA Updater sends the last message to terminate the Application Upgrade Process. The OTA Update Receiver again acknowledges this by sending an appropriate reply and soon after that makes a switch to the new Application Code. The switch to the new application code was verified by observing the behavior of the application code running in the sensor node with the OTA Update Receiver type bootloader.
7. Future work

The concept explained in this report is aimed at realizing the core idea of it and was not intended to provide a solution without any consequence. During the development of this concept the aspects to which we did not put more weightage for the sake of simplicity are security and robustness. That being said, there is a certain level of security and robustness present inherently. For example, a certain level of error handling in the packet exchange is taken care in the IEEE 802.15.4 standard protocol. Hence, as a future development, the concept can be augmented with features such as security enhancement and error recovery during the faulty write processes in the sensor nodes. Also, the concept deals with dissemination of the new code to the sensor nodes with only a single hop. Hence, work carried out in [9] can be referenced as a model to develop a more capable concept wherein code dissemination can be done on a larger scale and in a heterogeneous network as well.
Over The Air programming of WSNs is important in facilitating their management and maintenance, as well as permitting adaptive sensor applications. It is impossible to predict all the problems that may arise while installing the sensor nodes before deploying a sensor network. Hence, Over The Air programming is necessary to fix bugs, update codes, and manage application requirement changes. It eliminates the necessity to detach the sensor nodes and attaching them to data transfer cables to update the sensor software.

Many solutions have been designed so far for reprogramming the sensor nodes. But almost all of these solutions are specific to a particular hardware or software platform they are developed for. Also, there is not much work observed in finding solutions for highly resource constrained sensor nodes. The proposed concept tries to address this issue. This report explains how bifurcation of the firmware of the sensor node can aid in its Application upgradation process. By using the de facto IEEE 802.15.4 standard protocol for the communication, the complexity of the network communication was kept minimal. The proposed concept has been validated positively by its successful implementation along with the relevant corroborating results.
REFERENCES


## Contents in C.D

<table>
<thead>
<tr>
<th>Folder Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>