1. Introduction

1.1. Motivation

Gas sensors can be used in a great number of situations [1], and, if the technology is applied correctly, can lead to better results than other types of sensors (for example in fire detection). Another aspect of gas sensors is that they are most of the time sensitive to more than 1 type of gas which makes them very versatile. The GASB gas sensor, one of the hundreds of types of gas sensors in existence, is the focus point of the project described in this document. Its primary function is that of a smoke and fire detector but other usages are possible. The GASB gas sensor contains integrated electronics that facilitate controlling. However, even in this advanced form it is far from being able to provide raw measurement data in a digital easy-to-use form (for example as data on the PC). It is because of this reason that, as any type of sensor for that matter, the GASB gas sensor needs to be used together with controlling electronics. Given the versatility of the sensor and the multitude of its uses the best way to control it is via attached microcontroller (because microcontrollers are relatively fast, have a lot of available peripherals and interfaces and can be reprogrammed at will). This is how the GASB Sensor Platform was born. It is a complete microcontroller based system that houses four GASB sensors and sends the measurement data to a PC.

In practice the 1-to-1 Platform-PC connection leads to the need of a PC for every GASB Board which, in turn, causes setups of measurements to be difficult. An easier data transfer system is thus desired, preferably using radio links instead of cables.

This is the goal of the GASB Extension Project. Having Sensor Platforms connect wirelessly to a PC implies an increased degree of portability which leads to another desired feature: battery operated systems. The second goal of the project is, thus, to improve power consumption on the targeted devices.

1.2. Project Description

The “GASB Sensor Platform Wireless Capability Extension and Power Consumption Optimization” Project was started in order to allow multiple GASB Sensor Platforms to be used in different measurement scenarios while having the capability of collecting all the measurement data in a centralized way to facilitate later model-based calculations. The task is to use the existing hardware with certain extensions such as RF chip and antenna and develop software that would allow multiple devices to be connected to a single computer. As having multiple cabled connections would be cumbersome, a radio link between the computer and all the devices is preferred. Since a battery-powered operation is desired, special attention has been given to low-power capabilities of the radio link.

The main goal of this project is to design and implement the software for the given hardware in such a way that the task mentioned above is achieved but also the solution can be adapted to other similar platforms. The desired result is to have a flexible software design that can be adapted to future hardware with ease.

2. System Overview

2.1. Hardware Description

2.1.1. GASB Gas Sensor

The GAS 85xyB gas sensor (short GASB) developed by Micronas is a hybrid Flip Chip setup using floating gate field effect transistors produced in CMOS technology [2]. Since CMOS technology enables low power operation GASB sensors are well suited for battery-operated, wireless applications. Due to the hybrid setup it serves as a versatile gas sensing platform adaptable to numerous applications [1].

One of the best suited applications for this sensor is the detection of fires, because in this application the sensor has to be operated with very low power consumption in the mW range over time [3]. Here conventional optical smoke detectors will be replaced by highly integrated gas sensors “sniffing” the fire. A complex mixture of various compounds can be detected in fire gases, but the gases which arise amongst humidity in the higher concentration levels are CO, NO_2 and CO_2. These gas components are used as main target gases. For the detection of CO in fire detectors, electrochemical cells are already used in commercial products and European standards exist for their qualification, but their range of application is restricted by lifetime and price.

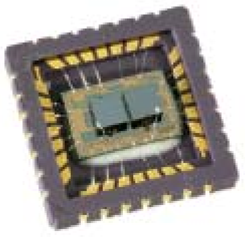


Figure 2-1: Floating Gate FET based gas sensor [4]

A second major application for GASB sensors is the reliable detection of CO_2 [3] as a key for demand controlled ventilation (DCV) in buildings, which allows energy savings up to 30%. Presently, optical detection methods are only in use for a small percentage of building ventilation systems due to the relatively high cost of these sensing systems. A reliable low cost CO_2 sensor would allow for widespread use of DCV.

2.1.2. GASB Sensor Platform

The GASB Sensor Platform is a sensor development and testing platform that provides all the hardware necessary to test and evaluate the GASB gas sensors. The GASB Platform houses a total of four GASB sensors and is able to execute commands such as measurement or calibration for each sensor individually. The commands are supplied via PC using a Serial-to-USB interface from a LabView program that both controls the four sensors and logs the data into graphs.

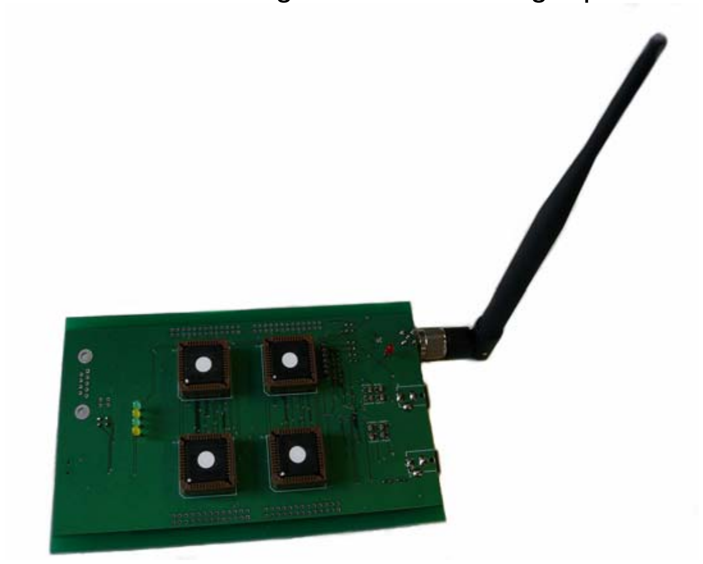


Figure 2-2: GASB Sensor Platform with RF Extension

The GASB Sensor Platform has at its core the Texas Instruments MSP430F1611 Microcontroller and it has been also extended with the Texas Instruments CC2500 RF Modem and an antenna. The combination of the two chips is able to run the Texas Instruments SimpliciTi RF protocol [6] in order to achieve radio communication. However, the MSP430F1611 belongs to a family that is slightly different than the MSPs used in most of the SimpliciTI examples and also does not have a Board Support Package readily available from TI. Therefore custom support drivers must be developed for the GASB Sensor Platform in order to be able to use SimpliciTi.

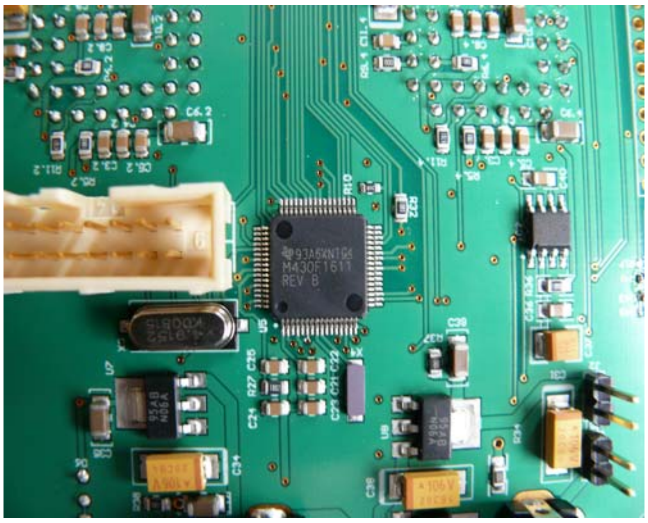


Figure 2-3: The MSP430F1611 mounted on the GASB PCB

The GASB Sensor Platform can be reprogrammed by making use of the JTAG port (seen in the picture above, to the left) and a Texas Instruments MSP-FET430UIF T programming tool.

2.1.3. EZ430-RF2500 Evaluation Kit

The eZ430-RF2500 is a complete USB-based MSP430 wireless development tool providing all the hardware and software to evaluate the MSP430F2274 microcontroller and CC2500 2.4-GHz wireless transceiver.

The eZ430-RF2500 uses the IAR Embedded Workbench Integrated Development Environment (IDE) or Code Composer Essentials (CCE) to write, download, and debug an application. The debugger is unobtrusive, allowing the user to run an application at full speed with both hardware breakpoints and single stepping available while consuming no extra hardware resources. The eZ430-RF2500T target board is an out-of-the box wireless system that may be used with the USB debugging interface, as a stand-alone system with or without external sensors, or may be incorporated into an existing design. The new USB debugging interface enables the eZ430-RF2500 to remotely send and receive data from a PC using the MSP430 Application UART.

eZ430-RF2500 features:

- USB debugging and programming interface featuring a driverless installation and application
- backchannel
- 21 available development pins
- Highly integrated, ultra-low-power MSP430 MCU with 16-MHz performance
- Two general-purpose digital I/O pins connected to green and red LEDs for visual feedback
- Interruptible push button for user feedback

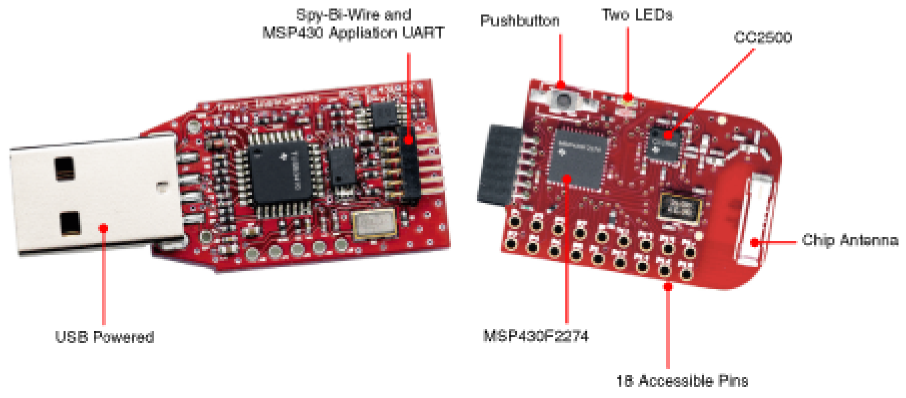


Figure 2-4: eZ430-RF2500 Evaluation Kit [5]

The eZ430-RF2500 Evaluation Kit contains two target boards and a programming tool (above only one of each). This provides the opportunity of testing networks with more than just an Access Point and an End Device if the same End Device program from the GASB Sensor Platform is running on one of the eZ430-RF2500 boards. Connecting GASB sensors to the eZ430-RF2500 would be difficult but for testing it is not needed as we are more interested in the network functionality so we can just feed dummy data to the Access Point.

Since the programming tool included in the kit also has the ability to bridge UART information from the RF module to the PC via Serial-to-USB, it will be used with the Access Point RF module at all times to facilitate data connection between the AP and the PC without the need of developing special hardware.

2.1.4. The CC2500

The CC2500 is a low-cost 2.4 GHz transceiver designed for very low-power wireless applications. The circuit is intended for the 2400-2483.5 MHz ISM (Industrial, scientific and Medical) and SRD (Short Range Device) frequency band. The RF transceiver is integrated with a highly configurable baseband modem. The modem supports various modulation formats and has a configurable data rate up to 500 kBaud. CC2500 provides extensive hardware support for packet handling, data buffering, burst transmissions, clear channel assessment, link quality indication, and wake-on-radio. The main operating parameters and the 64-byte transmit/receive FIFOs of CC2500 can be controlled via an SPI interface. In a typical system, the CC2500 will be used together with a microcontroller and a few additional passive components.

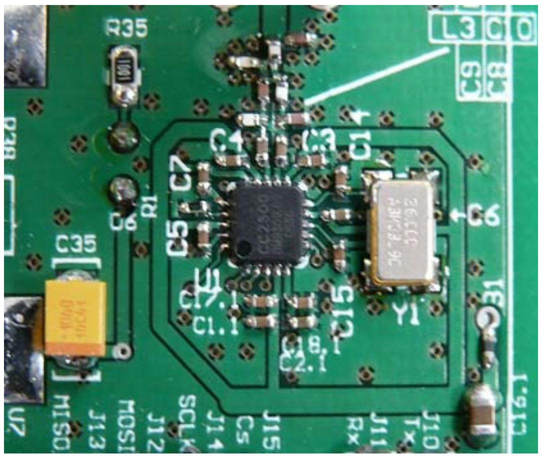


Figure 2-5: The CC2500 mounted on PCB

The features of the CC2500 that are essential for our application are enumerated below:

RF Performance

- High sensitivity (–104 dBm at 2.4 kBaud, 1% packet error rate)
- Low current consumption (13.3 mA in RX, 250 kBaud, input well above sensitivity limit)
- Frequency range: 2400 – 2483.5 MHz

Analog Features

- Suitable for frequency hopping and multichannel systems due to a fast settling frequency synthesizer with 90 us settling time
- Automatic Frequency Compensation (AFC) can be used to align the frequency synthesizer to the received centre frequency

Digital Features

- Digital RSSI output
- Support for automatic Clear Channel
- Assessment (CCA) before transmitting (for listen-before-talk systems)

Low-Power Features

- 400 nA SLEEP mode current consumption
- Fast startup time: 240 us from SLEEP to RX or TX mode (measured on EM design)
- Wake-on-radio functionality for automatic low-power RX polling
- Separate 64-byte RX and TX data FIFOs (enables burst mode data transmission)

General

- Few external components: Complete on-chip frequency synthesizer, no external filters or RF switch needed
- Small size (QLP 4x4 mm package, 20 pins)
- Support for asynchronous and synchronous serial receive/transmit mode for backwards compatibility with existing radio communication protocols

Aside from the SPI connection that interfaces the CC2500 with the microcontroller, there is a digital input that can be connected to an interrupt-able port on the microcontroller. The CC2500 will set this line high whenever a frame has been received. This connection is very important as it is used to trigger an Interrupt Service Routine from within the communication protocol.

2.2. Software Description

2.2.1. SimpliciTi: Low Power RF Network

SimpliciTI™ is a low-power RF protocol aimed at simple, small RF networks. This software is open-source and it provides an easy way to build a network of battery-operated devices using one of TI's low-power RF System-on-Chips (SoC) or the MSP430 ultra-low-power MCU and a TI RF transceiver.

SimpliciTI was designed for easy implementation and deployment out-of-the-box on several TI RF platforms such as the MSP430 MCUs and the CC1XXX/CC25XX transceivers and SoCs. SimpliciTI supports 2 basic topologies. The first is a Star topology which contains an Access Point as the hub and the second involves using default tokens for connecting different End Devices with one another (peer to peer).

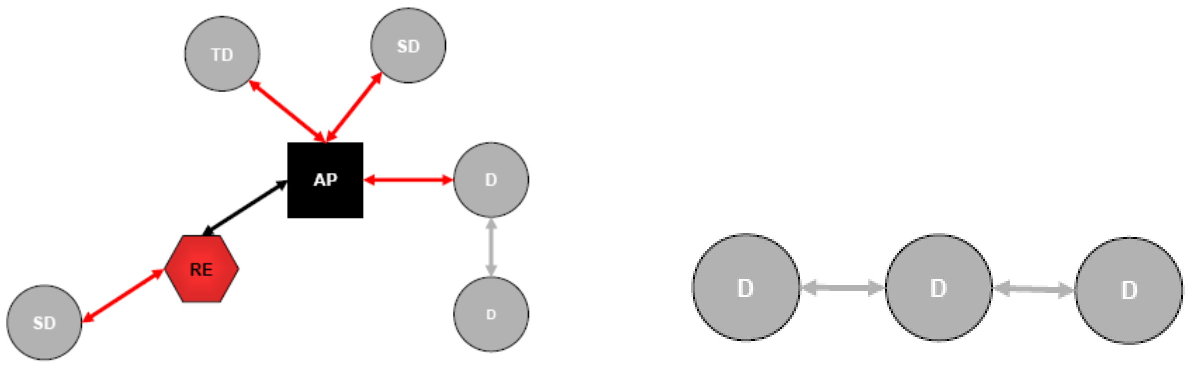
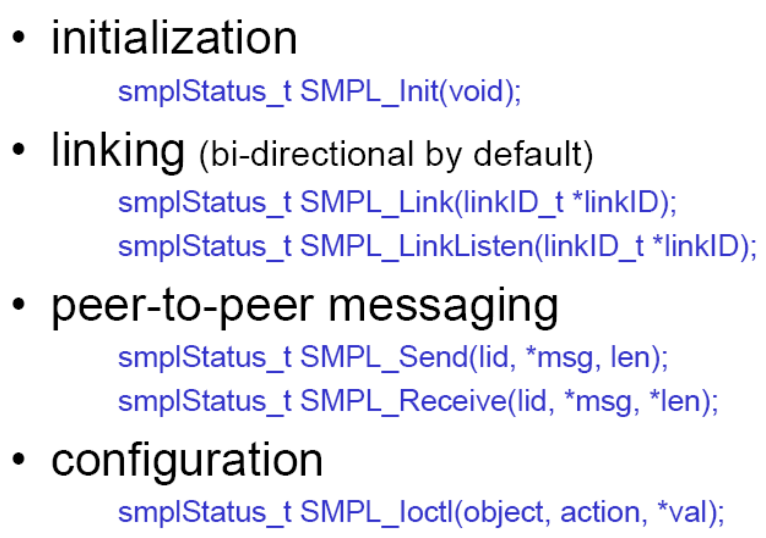


Figure 2-6: Example of SimpliciTi Star and Peer to Peer Network [5]

The previous picture shows the two main network topologies of SimpliciTi. For the Star network there are several types of End Devices such as repeaters (RE), devices that transmit and receive normally (D), sleeping devices (SD) and transmit only devices (TD).

The Access Point is used primarily for network management duties. It supports such features and functions as store-and-forward support for sleeping End Devices, management of network devices in terms of membership permissions, linking permissions, security keys, etc. the Access Point can also support End Device functionality, i.e., it can itself instantiate sensors or actuators in the network. The protocol support is realized in a small number of API calls.



- initialization
  smplStatus_t SMPL_Init(void);
- linking (bi-directional by default)
  smplStatus_t SMPL_Link(linkID_t *linkID);
  smplStatus_t SMPL_LinkListen(linkID_t *linkID);
- peer-to-peer messaging
  smplStatus_t SMPL_Send(lid, *msg, len);
  smplStatus_t SMPL_Receive(lid, *msg, *len);
- configuration
  smplStatus_t SMPL_Ioctl(object, action, *val);

Figure 2-7: SimpliciTi API

These APIs support Customer application peer-to-peer messaging. The association between two applications, called linking, is done at run time. The linking process creates a connection based object through which the application peers can send messages. When a connection is established it is a bi-directional connection. The End Devices can also have special features like Sleeping End Device or Transmit Only End Device. The network can be extended by use of Repeaters who simply repeat incoming messages (with limitations) and are completely transparent to the network otherwise.

2.2.2. User Application

The user application is in fact the main program that will run on the system. It is comprised of two parts: a program that will be implemented on the End Device boards and a program that will run on the Access Point. It should be mentioned that it is desired to implement the RF SimpliciTi functionality into more sensor platforms in the future, not just the GASB Sensor Platform so it would help if the End Device program is tailored to contain a base template that can easily be adapted to other platforms and solutions. The user application represents the main focus of this project.

The user application interfaces with SimpliciTi in the following way:

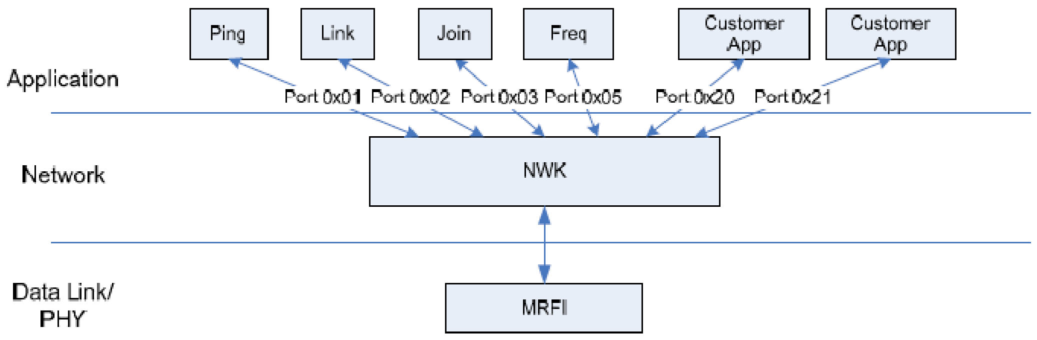


Figure 2-8: SimpliciTi Architectural Overview [5]

As show above the user application can use the SimpliciTi API (join, link, etc...) or implement new network functions using the underlying SimpliciTi protocol stack. For the purpose of this project the standard SimpliciTi API should suffice. One must keep in mind, however, that an extension to this API is possible and greatly increases the flexibility of the desired concept to design.

3. Design Considerations

3.1. Assumptions and Dependencies

3.1.1. Board Support Package

The EZ430-RF2500 Evaluation Kit has its own Board Support Package (BSP) which makes it directly usable with SimpliciTi. The GASB Sensor Platform however does not. Therefore a custom BSP was developed for the GASB Sensor Platform by using the EZ430-RF2500 one as example.

The most notable changes are to define the SPI interface with which the MSP controller communicates with the RF modem and to different pins and signals connections such as LEDs or the RF modem’s interrupt signal.

It should be noted that since the aim of the Board Support Packages is to present hardware functions to the user application in an abstract way, having BSP defined for each platform or device that is used helps implementing the same programs on different hardware configurations with little to no modifications in software.

3.1.2. Access Point and End Devices

Since the aim of the project is that the Access Point collects real time data from all available End Devices it is required that the Access Point is directly connected with the End Devices. In the network topology of the SimpliciTi protocol this means in fact that the device that will receive the Access Point functionality will have End Device capabilities aswell.

3.2. General Constraints

The use of the chosen hardware and as well the SimpliciTi protocol will impose some constraints on the whole system.

3.2.1. Hardware Constraints

SimpliciTi requires RAM to allocate whenever opening a new connection. Given the fact that the Access Point and End Devices are implemented on microcontrollers with limited RAM the default upper limit of 30 simultaneous connections for SimpliciTi will become a fixed limitation. This translates to the fact that the Access Point will be able to connect to a maximum of 30 Sensor Platforms. This represents no problem for the planned initial tasks for the sensor network system but may become one in the future. One way to solve this is to use End Devices as network hubs. Because all they will do is forward messages to the Access Point, their hardware and software structure should be relatively simple, however if each of the 30 connections of the AP is taken by a hub or node and then each node’s remaining 29 connections are taken by End Devices with sensors then one can achieve a maximum of 29*30=870 connections. Moreover, using this solution to expand the network may not even cause data transfer to slow down if one considers the possibility of having the nodes compress multiple End Device frames into one and then forward it to the Access Point. If this should still not be enough one can apply another level of network nodes. For this project however the default 30 connections limit will suffice.

3.2.2. Network Topology Constraints

Another constraint imposed by the chosen network topology (Star network) is that the Access Point can only handle synchronization with one End Device at a time. Because the End Devices are mostly in low power mode the Access Point cannot tell when they have woken up so the End Devices must send a message on each wake up to signal the Access Point. Once the message is received the Access Point has a small time window to send back the command message and receive the outcome of the command's execution. This means that any other End Devices that are trying to notify the Access Point they have awaken from low power mode during this time window must and will be ignored. As far as the internal mechanics of SimpliciTi go there is no way to receive and send frames simultaneously from the same peer (full duplex) nor is there a way to send two different frames to two different peers in the same time. Receiving more frames at once from multiple peers is also not possible, however the odds that they arrive in the same time are minimal and even a small time difference will ensure that they both end up in the received frame queue. The problem that arises in this case is that if an acknowledgement is required for the received frames, the Access Point will probably not be able to send both acknowledgements in sufficient time. A way to quickly solve this issue is to have the End Devices send synchronization frames periodically with a large enough interval so that the chances that two of them will send at the same time are negligible.

3.3. Goals and Guidelines

The main goal of the project is to achieve radio connection between a PC and multiple Sensor Platforms in order to receive real-time measurement data. The PC will be able to receive and transmit RF frames with the help of an Access Point device that is attached via USB and is able to communicate with the PC via Serial-to-USB and as well with the End Devices via its RF Modem. From the available network topologies of SimpliciTi, the one that suits these needs the best is the Star Network. In the Star Network however the Access Point handles only network management such as joining and forwarding frames, while the peers link themselves to other peers. This aspect is undesired in our solution, so the Star network topology will be customized. The

Access Point device will receive the ability to link to End Devices thus gaining End Device capabilities itself.

Because one of the goals is to develop a base template that uses SimpliciTi and only handles the connection and data transfer part of the program so that this template can later be ported to other sensor platforms the software to be developed will focus on those aspects. Since real-time measurement is desired but the result of the measurement may arrive at the Access Point and then on the computer at a later point in time given the network transfer delays, the following idea seems practical for a first implementation:

- *The Access Point will have its own clock that will provide for a system time*
- *The End devices will also have a clock that drives their own system time and will report this value on each synchronization frame*
- *The Access Point will reply with it's own system time (which should be the most accurate since it doesn't go into low power mode at all) and optionally with a command that depends on the later usage of this application*
- *The End Device will store the received time and use it with their internal clock while entering low power mode*

Next time the End Device wakes up it will send in the synchronization frame a time value that represents the time received from the Access Point plus the delay measured by the End Device itself. By comparing these two values one can tweak the different required delays in communication to ensure that all devices have the same notion of time. This means it does not matter when exactly the measurement result arrives at the computer as it will come together with the time when the measurement was taken.

3.4. Development Methods

The programming will be done in C language using Texas Instruments' Code Composer Studio 4 as IDE. For debugging, the hardware debuggers can be used to step through programs and check internal mechanics, and as for real time functionality (since stepping through the program of two different devices in the same time is impossible with a single computer) a terminal program will be used to display text messages with runtime information received on the PC through the Serial-To-USB interfaces of each device.

4. Architectural Strategies

4.1. Code Composer Studio 4

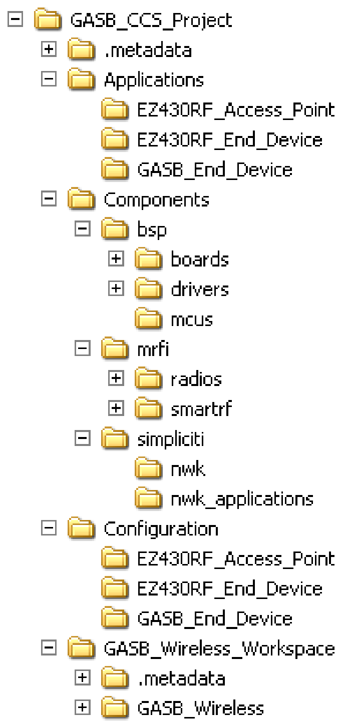
Code Composer Studio 4 (CCS4) was used for code development and debugging. Since all the microcontrollers in use belong to TI MSP families it makes sense to use their recommended IDEs. In comparison to the IAR Compiler which has a built in limitation of 4kb program size, Code Composer Studio 4 has a limit of 16kb. This makes Code Composer Studio 4 the better choice for development as using the SimpliciTi API will increase program size considerably. Other advantages of using Code Composer Studio 4 reside in the fact that CCS4 is based on the Eclipse IDE which provides many tools for code creation and maintenance.

4.2. CCS Project Structure

CCS4 allows projects with sophisticated file structures to be built. This advantage will be exploited in order to provide a single project that should contain all the necessary resources to implement programs on both of our platforms.

4.2.1. File Structure

Since one of the goals is ease of porting to other platforms it makes sense to separate the actual project folder from the different resources used by it.



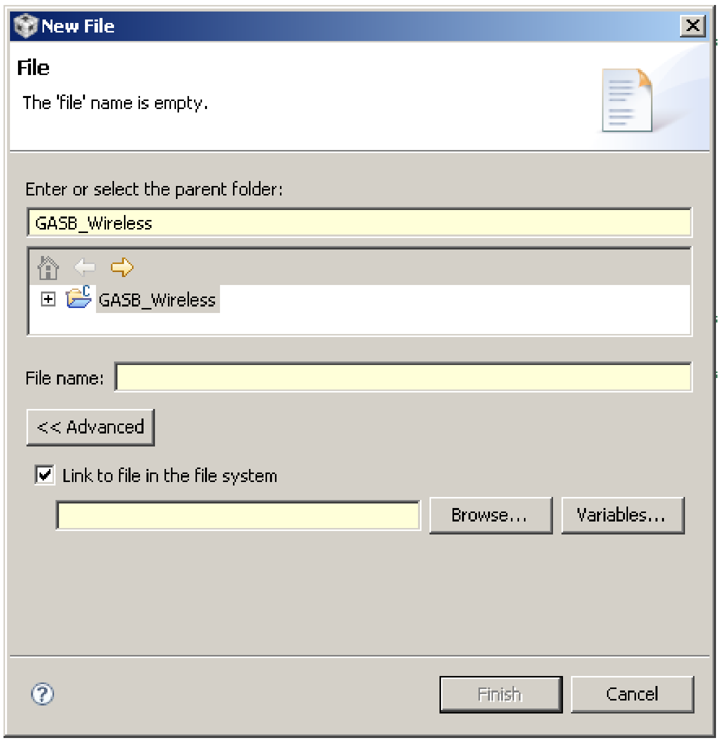
```
⊟ GASB_CCS_Project
    ⊞ .metadata
    ⊟ Applications
          EZ430RF_Access_Point
          EZ430RF_End_Device
          GASB_End_Device
    ⊟ Components
        ⊟ bsp
            ⊞ boards
            ⊞ drivers
              mcus
        ⊟ mrfi
            ⊞ radios
            ⊞ smartrf
        ⊟ simpliciti
              nwk
              nwk_applications
    ⊟ Configuration
          EZ430RF_Access_Point
          EZ430RF_End_Device
          GASB_End_Device
    ⊟ GASB_Wireless_Workspace
        ⊞ .metadata
        ⊞ GASB_Wireless
```

Figure 4-1: Code Composer Project File Structure

As it can be seen in the picture above, the project folder is under \\GASB_CCS_Project\GASB_Wireless_Workspace\GASB_Wireless. In order to load the project in CCS4 one must first open the workspace in \\GASB_CCS_PROJECT\GASB_WIRELESS_WORKSPACE. The actual source files for the applications are to be placed in the \\GASB_CCS_PROJECT\Applications folder and should be linked from there. The same goes for the SimpliciTi components found under \\GASB_CCS_Project\Components.

4.2.2. Linked Resources

SimpliciTi has a complicated file structure that is very hard to adapt or include to projects. The easiest way to add SimpliciTi to a project is by linking certain files into the project, however, the SimpliciTi source files cannot be in the project's workspace or they will be compiled as objects independent of the include rules and the linker will report duplicates. Code Composer Studio 4 has the ability to link files outside the workspace to a project.


New File

File

The 'file' name is empty.

Enter or select the parent folder:

GASB_Wireless

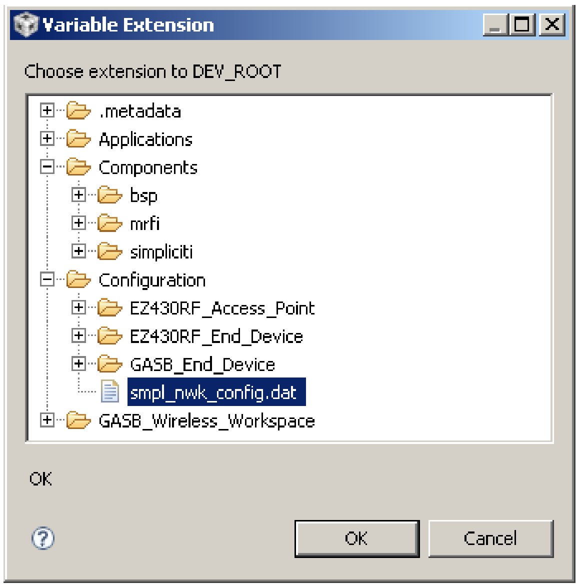
GASB_Wireless

File name:

Link to file in the file system [✓]

Figure 4-2: Linking a file to a CCS4 Project

However, if an exact path is given then the project will not be portable anymore (which means it must always be placed in a fixed location on each computer it will be used). This is not desired so a different approach was used.


Variable Extension

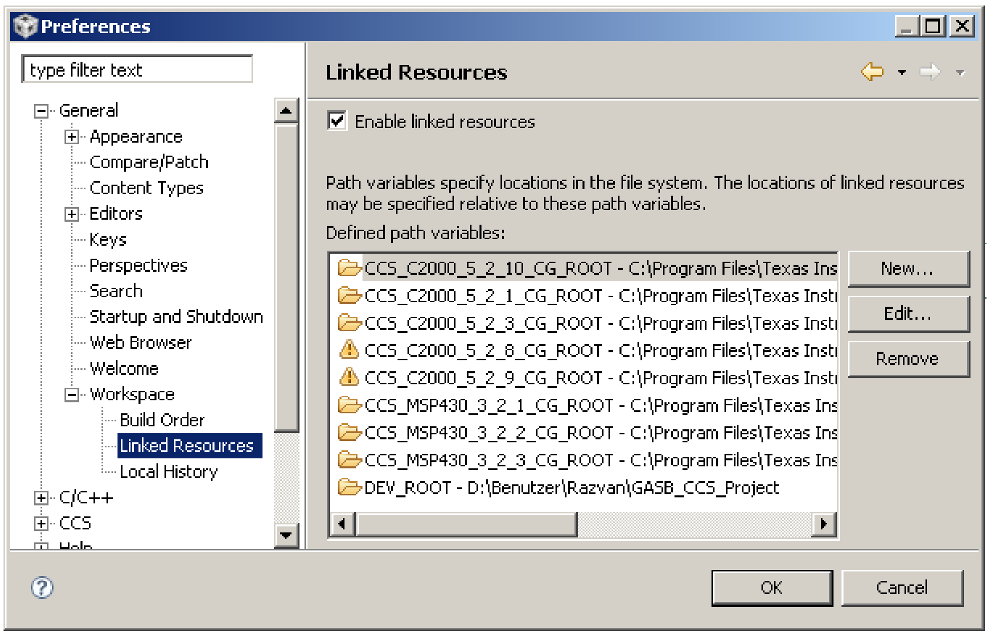
Choose extension to DEV_ROOT

```
.metadata
Applications
Components
    bsp
    mrfi
    simpliciti
Configuration
    EZ430RF_Access_Point
    EZ430RF_End_Device
    GASB_End_Device
    smpl_nwk_config.dat
GASB_Wireless_Workspace
```

OK

Figure 4-3: Linking files using variable paths

By defining a path as a variable we can create all file links based on that single variable. This means that if the project is moved to another computer in a different location, all that needs to be changed is the path variable called DEV_ROOT.


Preferences

type filter text

```
General
    Appearance
    Compare/Patch
    Content Types
    Editors
    Keys
    Perspectives
    Search
    Startup and Shutdown
    Web Browser
    Welcome
    Workspace
        Build Order
        Linked Resources
        Local History
C/C++
CCS
Help
```

Linked Resources

[✓] Enable linked resources

Path variables specify locations in the file system. The locations of linked resources may be specified relative to these path variables.

Defined path variables:

```
CCS_C2000_5_2_10_CG_ROOT - C:\Program Files\Texas Ins
CCS_C2000_5_2_1_CG_ROOT - C:\Program Files\Texas Insti
CCS_C2000_5_2_3_CG_ROOT - C:\Program Files\Texas Insti
CCS_C2000_5_2_8_CG_ROOT - C:\Program Files\Texas Insti
CCS_C2000_5_2_9_CG_ROOT - C:\Program Files\Texas Insti
CCS_MSP430_3_2_1_CG_ROOT - C:\Program Files\Texas Ins
CCS_MSP430_3_2_2_CG_ROOT - C:\Program Files\Texas Ins
CCS_MSP430_3_2_3_CG_ROOT - C:\Program Files\Texas Ins
DEV_ROOT - D:\Benutzer\Razvan\GASB_CCS_Project
```

Figure 4-4: Changing a path variable in CCS4

As shown in the picture above, the DEV_ROOT variable can be changed to the new project location and all the linked resources will be updated to the correct paths.

It goes without saying that setting up a project in a way that makes it easy for more people to use and removing all dependencies to a specific computer or environment speeds up development by a great amount.

4.2.3. Applications

The reason the application source files have been added as linked resources to the project structure as well is the following: if the components used are common for multiple platforms running multiple programs, then the actual programs can also be made into cross-platform compatible resources. With the current structure, one simply needs to define a new project in the workspace (or, even better, a new configuration in the current project), link the appropriate resources such as application to be used and SimpliciTi configuration and only care about the platform specific definitions in the Board Support Package. Once that is done all that is needed is to compile and upload to target board. With all options correctly set up, choosing X type of network device with the Y program variant on the Z target board where X,Y and Z can be any of the available resources in the respective categories, should take less time than the compile and upload process (not more than a minute).

One thing to mention is that one must care for this aspect when writing new applications. Since the Board Suport Packages are a fixed requirement of SimpliciTi, a feature they provide can be used. One of the definitions in the BSP of a board is the board's name. This can be used to choose the specific parts of the application that apply only to a certain target board automatically at compile time.

For example:

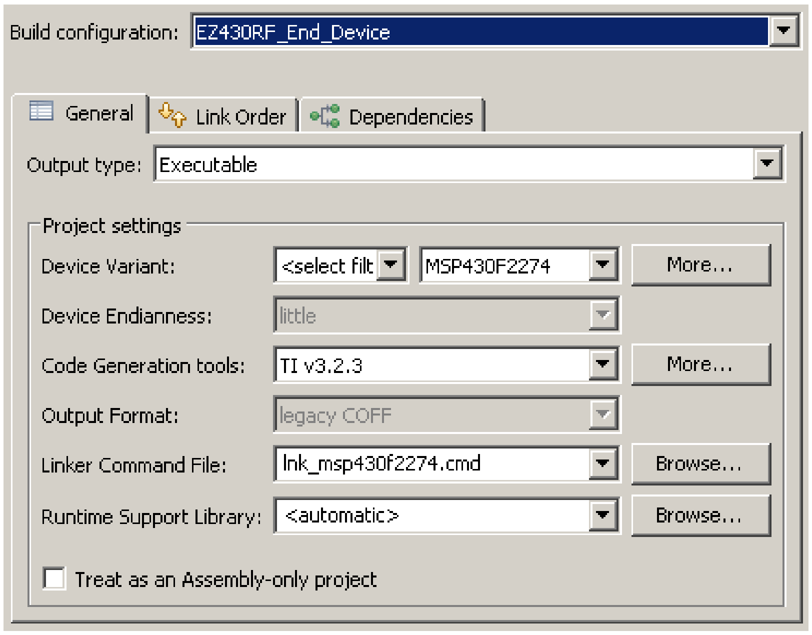
```
#if defined(BSP_BOARD_EZ430RF)
#pragma vector=TIMERB0_VECTOR
__interrupt void TB0_ISR (void)
{
      systemTimeID++;
      systemTimer++;
      if(systemSleep == 1){
            systemSleep = 0;
      }
      else{
            __bic_SR_register_on_exit(LPM3_bits);
      }
}
#elif defined(BSP_BOARD_GASB)
#pragma vector=WDT_VECTOR
__interrupt void WDT_ISR(void)
{
      systemTimeID++;
      systemTimer++;
      if(systemSleep == 1){
            systemSleep = 0;
      }
      else{
            __bic_SR_register_on_exit(LPM3_bits);
      }
}
#endif
```

As shown above, depending on which target board is used in the active configuration of the project, the system timer will be driven by either Timer B in case of the EZ430RF or the Watchdog Timer in case of the GASB board. It should be noted that such code constructions do not lower performance as the choice is made at compile time and the code that was not chosen is simply discarded and unused.

4.2.4. Project Configurations

Project configurations allow a single project with a great number of linked resources such as components or applications to produce different programs (by using different combinations of components and applications from the ones available) and also build them for different target boards. This feature is essential when considering cross-platform software.

As a short example, the following set of pictures will show how easy it is to change between programs or platforms.



Build configuration: EZ430RF_End_Device

Output type: Executable

Project settings

Device Variant: <select filt MSP430F2274

Device Endianness: little

Code Generation tools: TI v3.2.3

Output Format: legacy COFF

Linker Command File: lnk_msp430f2274.cmd

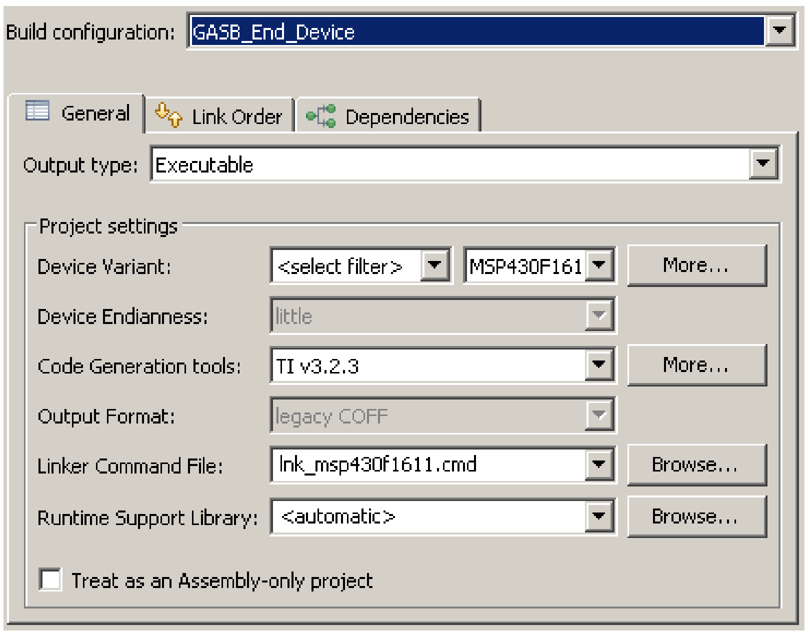
Runtime Support Library: <automatic>

☐ Treat as an Assembly-only project

Figure 4-5: Project configuration for EZ430RF as ED

The picture above shows that the target platform has been chosen as the EZ430RF and the application as that of an End Device.

Using this configuration a copy can be created and then the target device options can be changed. The new configuration will be the same as the old one but will apply to a different device.



Build configuration: GASB_End_Device

Output type: Executable

Project settings

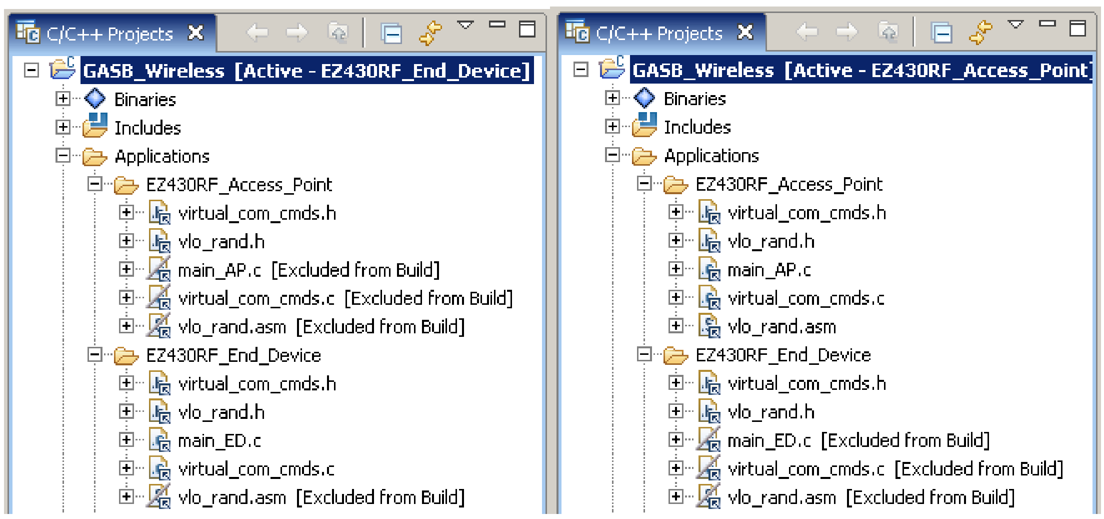
Device Variant:	<select filter> MSP430F161	More...
Device Endianness:	little	
Code Generation tools:	TI v3.2.3	More...
Output Format:	legacy COFF	
Linker Command File:	lnk_msp430f1611.cmd	Browse...
Runtime Support Library:	<automatic>	Browse...

☐ Treat as an Assembly-only project

Figure 4-6: Project configuration for the GASB Platform as ED

The new configuration allows the GASB Sensor Platform to be used as End Device. If special care was taken in setting up the resources file structure and definitions there should be no errors in trying to recompile the application for the GASB board.

Changing applications or components for the same device is even easier. Again, a copy of the specific configuration has to be made, but afterwards in order to choose a different application one must simply define which files will be excluded from build and which not.



```
C/C++ Projects
GASB_Wireless [Active - EZ430RF_End_Device]
  Binaries
  Includes
  Applications
    EZ430RF_Access_Point
      virtual_com_cmds.h
      vlo_rand.h
      main_AP.c [Excluded from Build]
      virtual_com_cmds.c [Excluded from Build]
      vlo_rand.asm [Excluded from Build]
    EZ430RF_End_Device
      virtual_com_cmds.h
      vlo_rand.h
      main_ED.c
      virtual_com_cmds.c
      vlo_rand.asm [Excluded from Build]
```

```
C/C++ Projects
GASB_Wireless [Active - EZ430RF_Access_Point]
  Binaries
  Includes
  Applications
    EZ430RF_Access_Point
      virtual_com_cmds.h
      vlo_rand.h
      main_AP.c
      virtual_com_cmds.c
      vlo_rand.asm
    EZ430RF_End_Device
      virtual_com_cmds.h
      vlo_rand.h
      main_ED.c [Excluded from Build]
      virtual_com_cmds.c [Excluded from Build]
      vlo_rand.asm [Excluded from Build]
```

Figure 4-7: Changing applications using project configurations

In the example above, the EZ430RF has two defined configurations, one as Access Point and one as End Device. For each of them there is a list of files to exclude so in either case the complete program that results after compilation will be a different one.

4.3. Debug methods

4.3.1. The CCS Debugger

Coupled with the debugging tools, the Code Composer Studio debugger can help stepping through a program, setting key breakpoints and also viewing variable and register contents.

In the early stages of program development it can be successfully used to check if register settings and algorithms are correct. This method is not very good in checking program functionality when testing radio communication as all triggers come from interrupts and some of the waiting procedures involve entering low power modes which in turn remove the ability of the programming tool to keep track of what the MSP is doing.

4.3.2. Serial-to-USB Interface

Since all the used hardware has the ability to send text data to a PC via a Serial-to-USB interface and the transmission itself, if carefully planned, does not disrupt time critical sections of the tested programs, one can use this feature in order to perform more accurate testing compared to using the CCS Debugger. This method allows for real-time testing and also gives the possibility of viewing output information from more than one device in the same time (essential in testing network connectivity and reliability).

5. System Architecture

5.1. System Functionality Overview

The following flow-chart shows the planned network traffic between the Access Point and at least two End Devices.

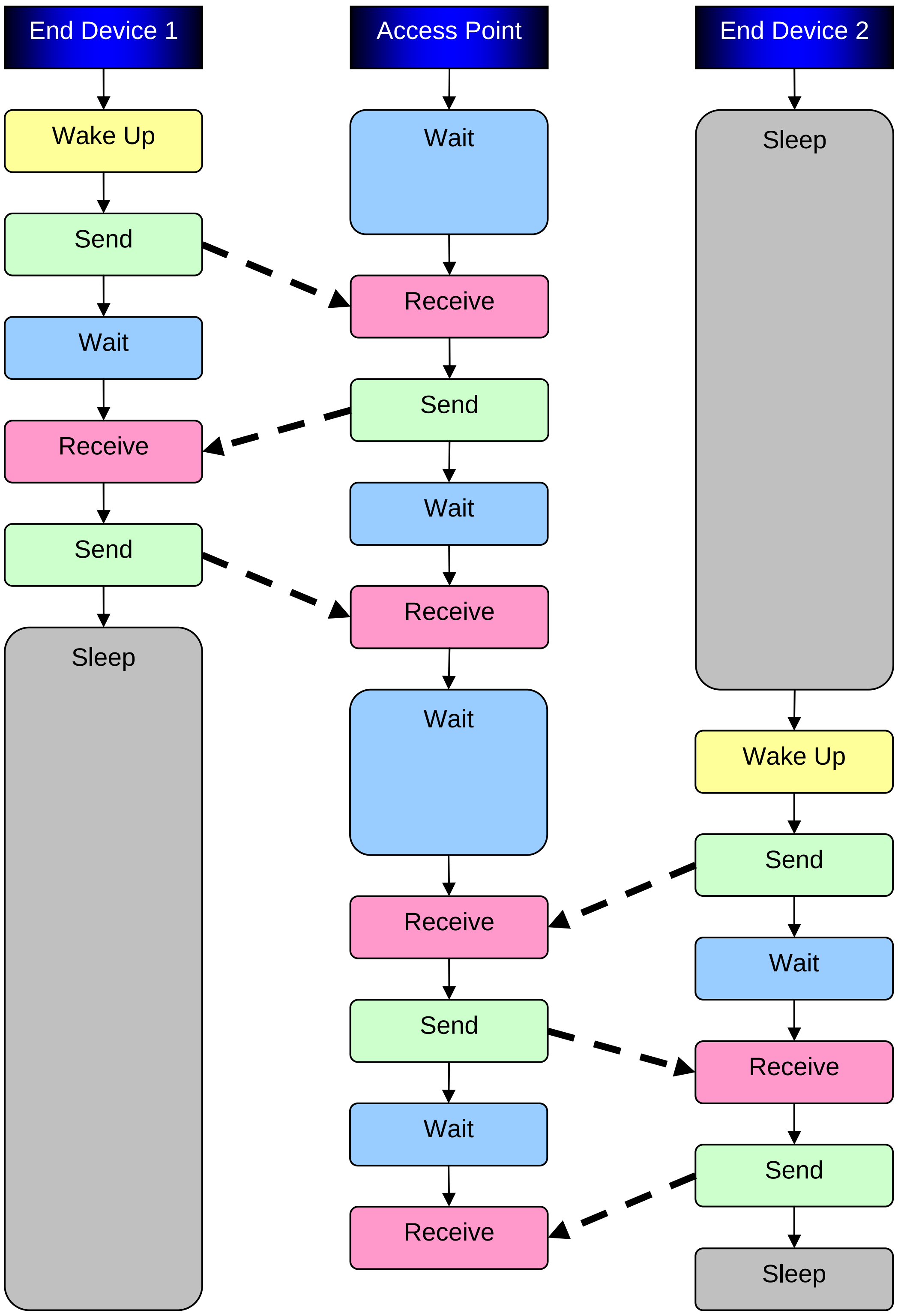


Figure 5-1: System connectivity

In the previous flow chart the dotted arrows represent radio communications. The main objective is to let the End Devices transmit to the Access Point when they come out of low power mode so that the Access Point can initiate a command/result transfer. Unlike presented in the flow chart in real life attempts End Devices may make an attempt to synchronize while the Access Point is already busy with another End Device. To prevent this, the exchange time has to be kept short and the Access Point must ignore extra requests when a transfer is underway.

5.2. Access Point Program Flow

The Access Point must be ready to answer End Device synchronization requests at all times as it cannot tell when End Devices come out of low power mode. The basic program flow is presented in the flowchart below. Please note that for a decision block the right choice represents "true" and the down choice represents "false". The dotted line at the end of the interrupt path means that the loop is conditioned by outside triggers (implied by an interrupts definition).

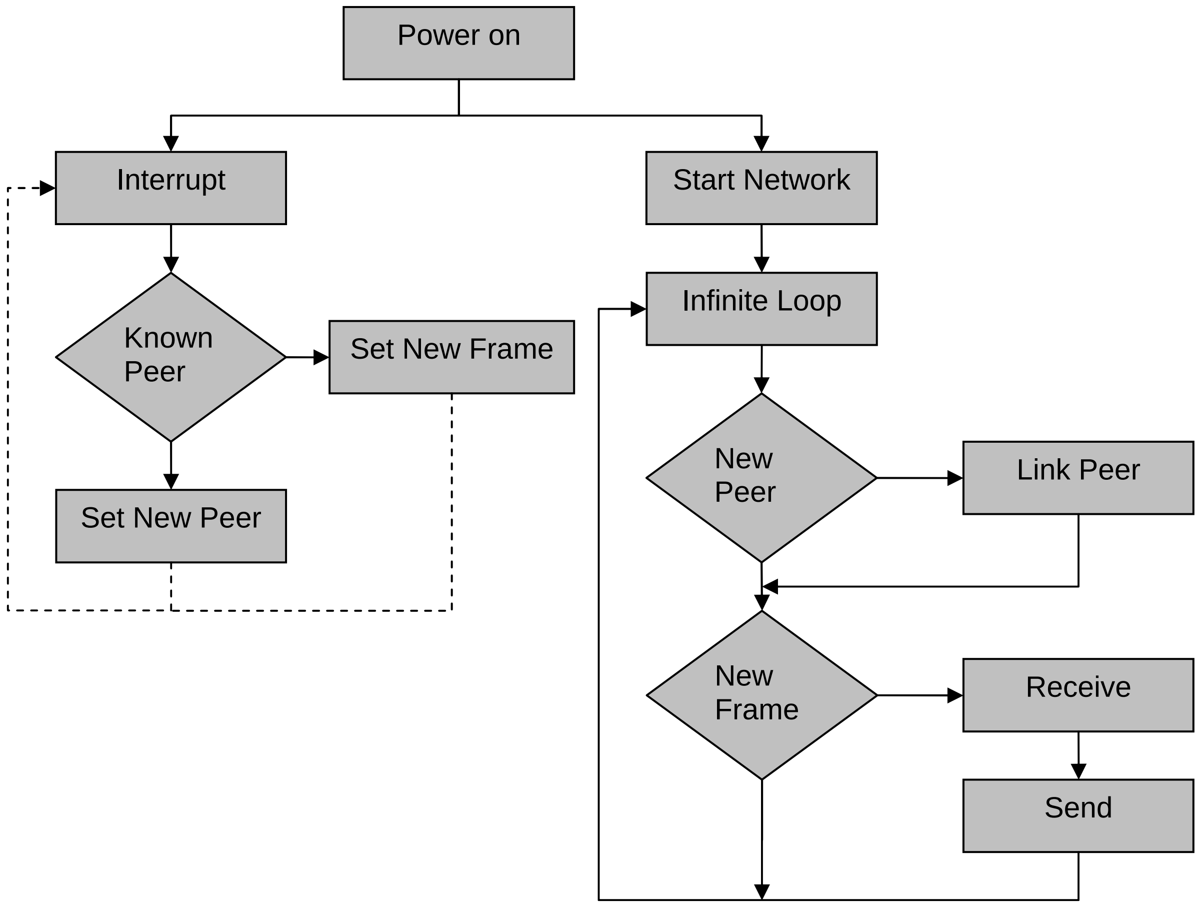


Figure 5-2: Access Point Program Flow

The flowchart above has been over-simplified but it still holds the most important information: the main loop of the program will take care of linking peers, reading frames and sending replies based on semaphores set by the SimpliciTi Interrupt Service Routine which can distinguish between a new peer that attempts to connect and an already connected peer that has just sent a frame to the AP.

5.3. End Device Program Flow

End Devices attempt to minimize the time they spend out of low power mode so they try to synchronize periodically with the Access Point on wake up and wait a short time for a reply.

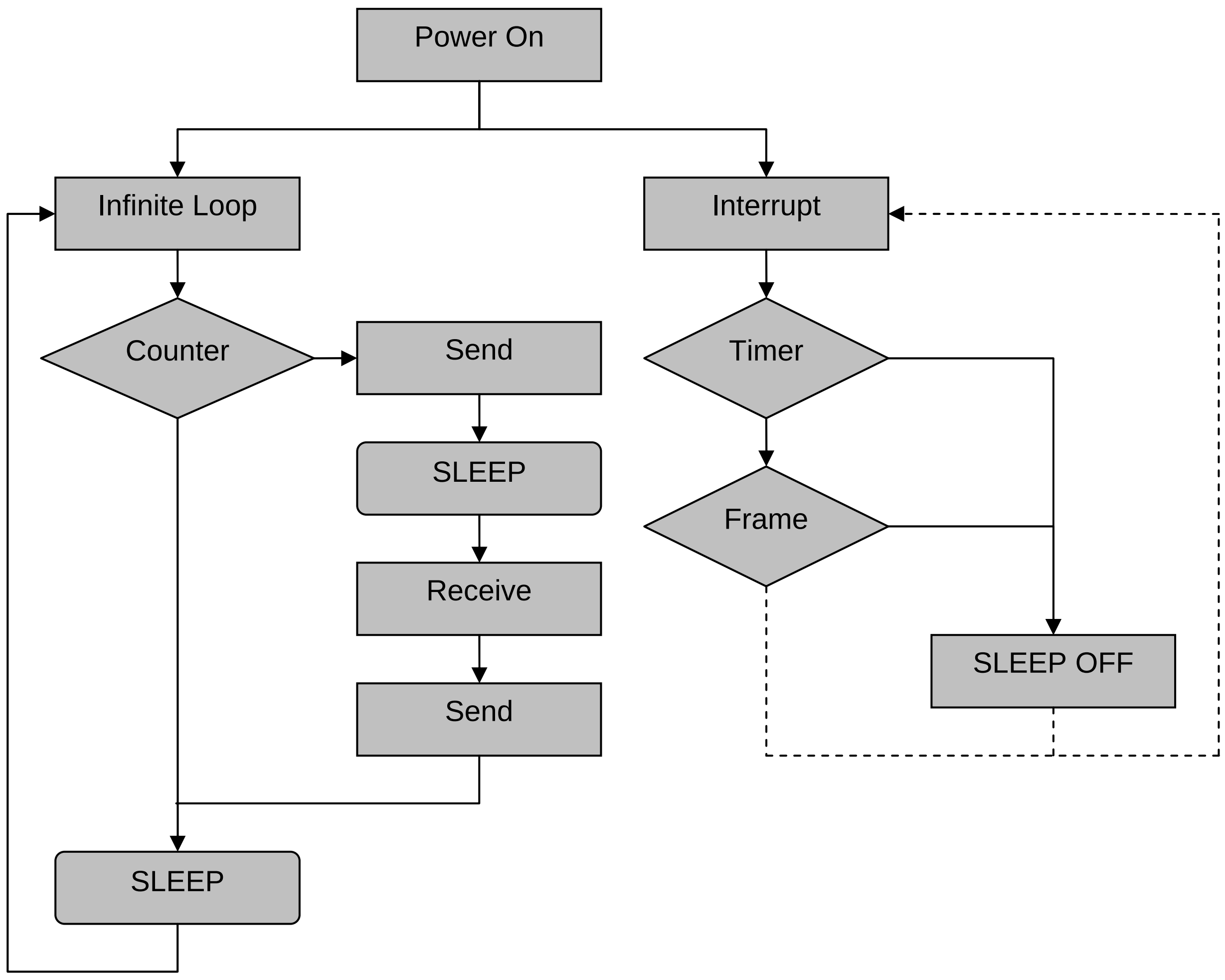


Figure 5-3: End Device Program Flow

The End Device has the ability to stop program flow and enter low power (or sleep) mode until an interrupt wakes it up again. Waking up from sleep is done in case a frame has been received or the system timer has sent another interrupt (once every second).

5.4. Other System Devices

Aside from the Access Point and the End Devices, other types of devices can be implemented such as Repeaters or Network Nodes (basically End Device for an AP that has its own connected End Devices and works as a gateway). This is not required nor is it planned for the current project but since it may become a valuable advantage to extend the network with such devices, all measures must be taken to facilitate any eventual implementation of said devices into the existing network. A simple example from Texas Instruments can prove the complexity level that can be achieved using multiple types of devices in a SimpliciTi network:

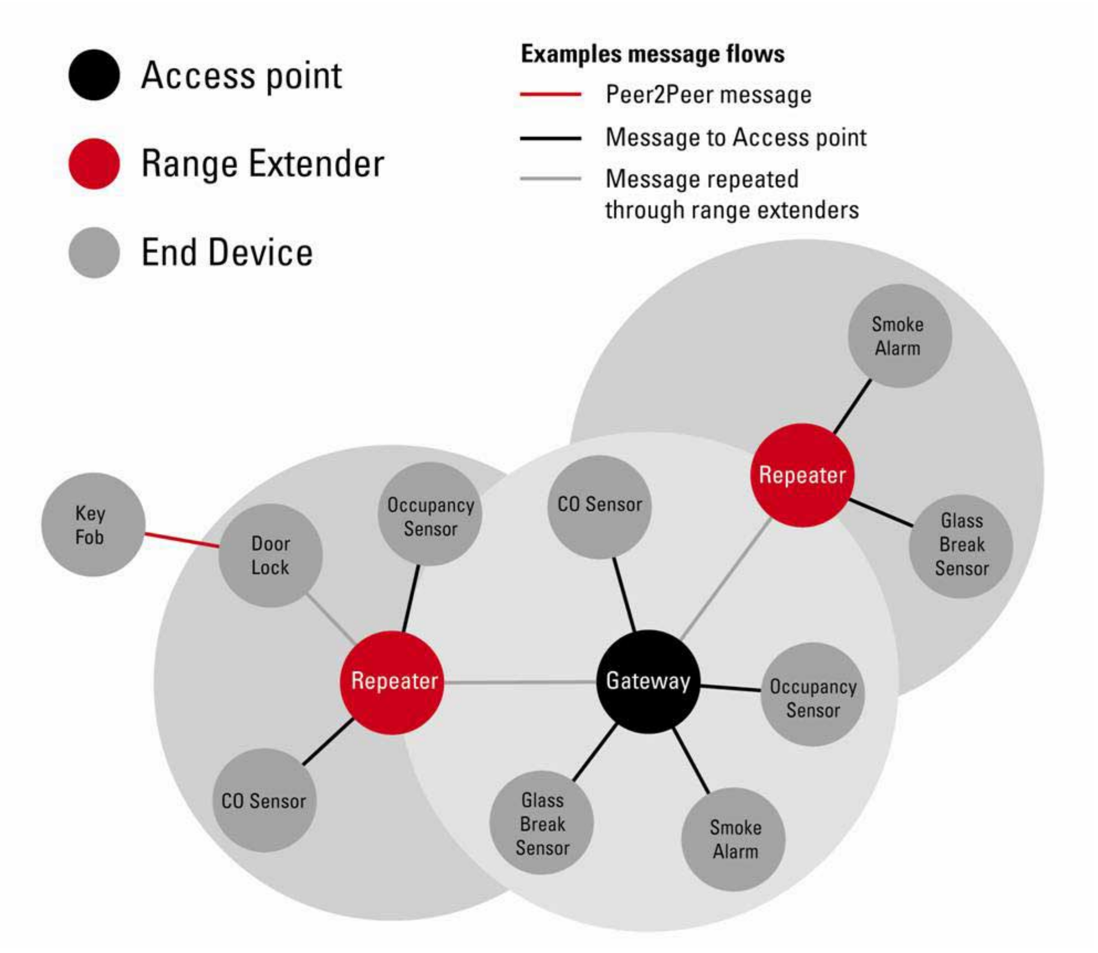


Figure 5-4: SimpliciTi Network Example [5]

It should be noted that Texas Instruments' concept of the Range Extender does not solve the problem of limited ports and that the before mentioned network node is, in fact, different in concept. It is the author's belief that this concept is easy to implement using the SimpliciTi protocol.

6. Policies and Tactics

6.1. Coding Guidelines

In order to improve the readability of the created code several conventions have been set in place:

- Variables and functions are to be named in a distinctive way starting with small letters and each subsequent word beginning with a capital letter.
- Macros are to be named using capital letters and each subsequent word should be separated by an underscore.
- If/else statement branches are to always contain an underlying block even if they lead to a single statement.
- Constants are to be defined as macros and they should be placed in an easily accessible section of the code (e.g. near the top of the file) separated from the rest of the variable and prototype declaration. This facilitates changing program parameters.
- Flags and switches must be defined using meaningful names. In the case a collection of flags is compressed bitwise into a single variable macros must be defined to address the respective flag in a descriptive way.

Another aspect worth mentioning is that using function calls in other function calls' parameter list is allowed even though it works against code readability because this method saves the use of an intermediary variable.

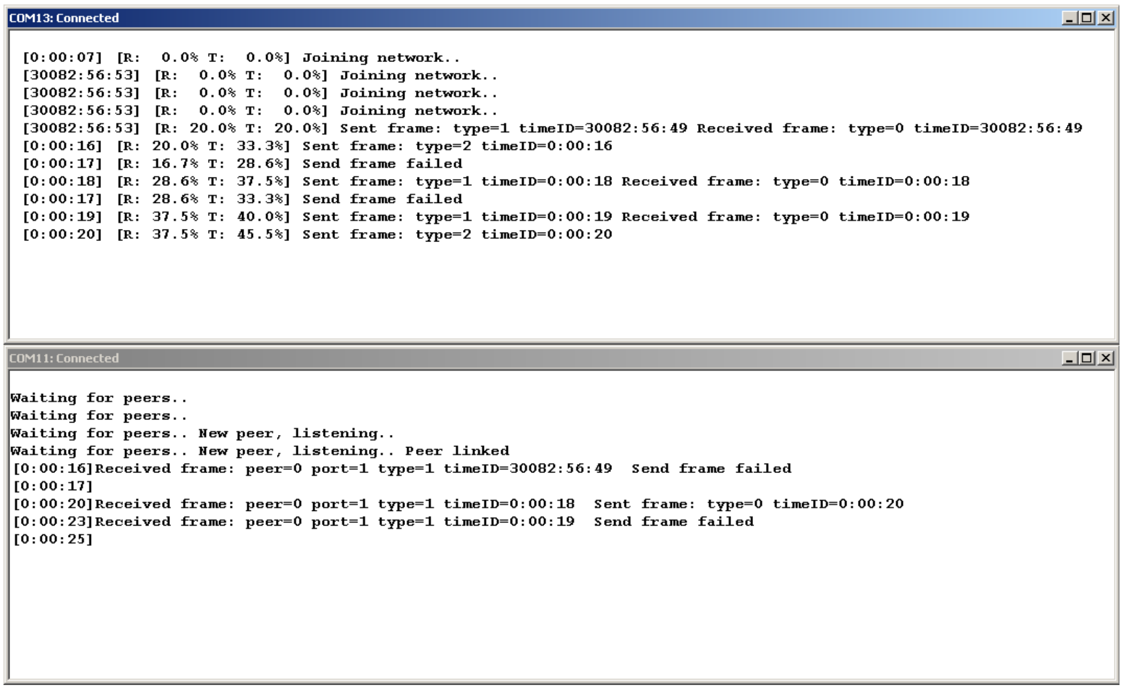
Finally, the use of local variables is discouraged with the exception of internal function variables that are defined as static. Global variables have the advantage that they are initialized only once and are accessible from all function scopes. The only other reason against using global variables is when a module type of behavior is desired for a certain function in order to improve portability and reusability.

6.2. Testing

Implementation of extra functions and variables in order to measure different test results is allowed as long as the removal of these additions is not complicated and they do not interfere with the main program.

6.3. Communication Interface

The Serial-to-USB interfaces of the used platforms were used to provide real-time data output in text form. The example below shows the output for an End Device communicating with the Access Point.



COM13: Connected

```
[0:00:07] [R:  0.0% T:  0.0%] Joining network..
[30082:56:53] [R:  0.0% T:  0.0%] Joining network..
[30082:56:53] [R:  0.0% T:  0.0%] Joining network..
[30082:56:53] [R:  0.0% T:  0.0%] Joining network..
[30082:56:53] [R: 20.0% T: 20.0%] Sent frame: type=1 timeID=30082:56:49 Received frame: type=0 timeID=30082:56:49
[0:00:16] [R: 20.0% T: 33.3%] Sent frame: type=2 timeID=0:00:16
[0:00:17] [R: 16.7% T: 28.6%] Send frame failed
[0:00:18] [R: 28.6% T: 37.5%] Sent frame: type=1 timeID=0:00:18 Received frame: type=0 timeID=0:00:18
[0:00:17] [R: 28.6% T: 33.3%] Send frame failed
[0:00:19] [R: 37.5% T: 40.0%] Sent frame: type=1 timeID=0:00:19 Received frame: type=0 timeID=0:00:19
[0:00:20] [R: 37.5% T: 45.5%] Sent frame: type=2 timeID=0:00:20
```

COM11: Connected

```
Waiting for peers..
Waiting for peers..
Waiting for peers.. New peer, listening..
Waiting for peers.. New peer, listening.. Peer linked
[0:00:16]Received frame: peer=0 port=1 type=1 timeID=30082:56:49  Send frame failed
[0:00:17]
[0:00:20]Received frame: peer=0 port=1 type=1 timeID=0:00:18  Sent frame: type=0 timeID=0:00:20
[0:00:23]Received frame: peer=0 port=1 type=1 timeID=0:00:19  Send frame failed
[0:00:25]
```

Figure 6-1:Example of debugging in real-time

Above the output of the Access Point (down) and End Device (up) is shown. Information on what network related task is performed is given together with contents of sent and received frames and transmit and receive success ratio for ED. The timestamp is the actual system time of each device.

7. Detailed System Design

7.1. Access Point Software Implementation

As previously mentioned the Access Point will have End Device capability such as linking and transmitting/receiving as well. At the time this software was written the main goal was to have a template that can also be transferred across multiple sensor platforms using SimpliciTi and not just be used for the GASB Sensor Platform. Because the Access Point would be used to connect all these devices together the software attempts to test the functionality of the radio communication by having the Access Point synchronize its system time with the End Devices. While just a test, this will still be useful later as, in case of measurements, the time the measurement was taken is required and not the time the Access Point received the result. Therefore we need to send the actual time to the End Devices and allow them to use this value for their internal clock and report back the time together with each measurement result.

7.1.1. Main Program Loop

The main program loop of the Access Point will call the appropriate SimpliciTi functions in order to retrieve data and send commands to the End Devices. In the future the received data will be sent to the computer via the serial interface and the commands will originate from the computer as well. The Access Point’s role is that of a gateway to a centralized control and data hub running on a PC.

The following section contains code snippets and explanation. Please note that the snippets put together constitute the whole function, thus explaining the different indentation used in each of them. They are to be seen as a whole which was broken and separated by text explanations.

First the Board Support Package, the serial interface and the system timer are initialized.

```
/********************************************************************\
  MAIN
\********************************************************************/

void main (void)
{
  /* Initialise */
  BSP_Init();
  COM_Init();
  START_TIMER();
  /* Start SimpliciTi network */
  LED_ALL_ON();
  SMPL_Status = SMPL_Init(sCB);
  LED_ALL_OFF();
```

The classic while(1) loop starts by checking the new peer semaphore and attempting to link to newly connected peers or attempts a link every loop run if there are no peers connected.

```
while(1){
  /* If new peer found or no peers connected listen for peers */
  if(newPeer > 0 || connectedPeers < 1){
    LED_ALL_ON();
    TXString(stringBuffer,sprintf(stringBuffer,"\r\n"));
    if(connectedPeers < 1){
      TXString(stringBuffer,
               sprintf(stringBuffer,"Waiting for peers.. "));
    }
    if(newPeer > 0){
      TXString(stringBuffer,
               sprintf(stringBuffer,"New peer, listening.. "));
    }
    SMPL_Status = SMPL_LinkListen(&sLID[connectedPeers]);
    if(SMPL_Status == SMPL_SUCCESS){
      newPeer--;
      TXString(stringBuffer,sprintf(stringBuffer,"Peer linked "));
      connectedPeers++;
    }
    LED_ALL_OFF();
  }
```

The new peer semaphore is incremented in the SimpliciTi callback function every time a new peer attempts to link. On each successful link this semaphore is decremented. In this way it is made clear that each of the End Devices that attempted to link will have their chance to catch the Access Point in listening mode.

Next the program enters another loop that executes until the frame semaphore becomes zero. Just like before, the frame semaphore is incremented on each received frame in the SimpliciTi callback function. On each successful read of the frame from the frame queue the semaphore is decremented.

```
  /* Try to read frames and reply until
                                frame semaphore is 0 again */
  while(newFrame > 0){
    LED_RED_ON();
    NWK_DELAY(BLINK_DELAY);
    LED_RED_OFF();
    /* try to receive frames for all peers */
    for(currentPeer = 0;
                currentPeer < connectedPeers; currentPeer++){
      SMPL_Status=SMPL_Receive(sLID[currentPeer],
                               receiveBuffer,
                               &receiveBufferlength);
      if(SMPL_Status == SMPL_SUCCESS){
        /* Print system time to the serial interface */
        TXString(stringBuffer,
                 sprintf(stringBuffer,
                         "\n\r[%u:%02u:%02u]",
                         TID_HOURS(systemTimeID),
                         TID_MINUTES(systemTimeID),
                         TID_SECONDS(systemTimeID)));
        newFrame--;
```

Because part of the received information is the End Device’s system clock the Access Point’s system clock is printed first in order to easily compare. This is also useful to mark the point in time when the message was received.

The first byte of the application data part of the message shows what kind of frame was received, whether it is a synchronization frame or the result frame following a command sent by the AP.

```
          /* If the frame received is a sync
                       frame reply with apropiate command */
          if(receiveBuffer[0] == 1){
            uint8_t tryCount = MAX_RETRIES;
            /* Decode received frame */
            frameTimeID=((uint32_t)receiveBuffer[1])+
                        (((uint32_t)receiveBuffer[2])<<8)+
                        (((uint32_t)receiveBuffer[3])<<16)+
                        (((uint32_t)receiveBuffer[4])<<24);
            TXString(stringBuffer,
                     sprintf(stringBuffer,
                             "Received frame: peer=%u port=%u
                               type=%u timeID=%u:%02u:%02u  ",
                             currentPeer,sLID[currentPeer],
                             receiveBuffer[0],TID_HOURS(frameTimeID),
                             TID_MINUTES(frameTimeID),
                             TID_SECONDS(frameTimeID)));
```

The AP also prints the End Device’s time to the serial interface in order to compare it to its own system time. It is known that the End Device’s system time comes from the last synchronization with the Access Point and we want to check if during the time spent sleeping the End Device measured time correctly.

Next we prepare the command frame to be sent and attempt to send it a number of times. If the send fails every time then a message is printed to the serial interface stating that the send failed. Depending on what the exact usages of this application will be when applied to a different system or situation, in this point one can implement a fail policy but in the present example no error handling is required as the Access Point is going to wait for the End Device to try again. Since the End Devices need to perform measurements once every few seconds over extended periods of time (in the range of days or weeks) it doesn’t matter if a small percentage of attempts fails.

```
          /* Prepare send frame */
          sendBuffer[0] = 0;
          sendBuffer[1] = (uint8_t)systemTimeID;
          sendBuffer[2] = (uint8_t)(systemTimeID>>8);
          sendBuffer[3] = (uint8_t)(systemTimeID>>16);
          sendBuffer[4] = (uint8_t)(systemTimeID>>24);
          frameTimeID = ((uint32_t)sendBuffer[1])+
                        (((uint32_t)sendBuffer[2])<<8)+
                        (((uint32_t)sendBuffer[3])<<16)+
                        (((uint32_t)sendBuffer[4])<<24);
          /* Try sending frame until successful or MAX_RETRIES */
          do{
            LED_GREEN_ON();
            SMPL_Status = SMPL_SendOpt(sLID[currentPeer],
                                       sendBuffer,
                                       sizeof(sendBuffer),
                                       SMPL_TXOPTION_ACKREQ);
            tryCount--;
            NWK_DELAY(BLINK_DELAY);
            LED_GREEN_OFF();
          }while(tryCount && SMPL_Status != SMPL_SUCCESS);
          if(SMPL_Status == SMPL_SUCCESS){
            TXString(stringBuffer,
                     sprintf(stringBuffer,
                           "Sent frame: type=%u
                                timeID=%u:%02u:%02u ",
                             sendBuffer[0],
                             TID_HOURS(frameTimeID),
                             TID_MINUTES(frameTimeID),
                             TID_SECONDS(frameTimeID)));
          }
          else{
            TXString(stringBuffer,
                     sprintf(stringBuffer,"Send frame failed "));
          }
        }
```

It should be noted that the frame sent by the Access Point in response to the synchronization frame received from an End Device has its own type (header or first byte of the application data in the frame). Also, while in this example we are only trying to synchronize time, there is always the possibility of adding more information to be sent such as a command. Using an extra byte would yield the possibility of sending a command encoded as an unsigned short integer which allows for 255 separate commands to be distinguishable by the End Device.

If the frame followed a result, in this example we don’t need to do anything but in the case of a measurement request that might be later implemented we must handle the incoming data.

```
          /* If the frame received was a result frame
                      process the information accordingly */
          if(receiveBuffer[0] == 2){
            //SPACE RESERVED FOR: result processing
          }
        }
      }
    }
```

Lastly in case the system timer (note: not system time) has reached the desired tick rate we reset it. Later when calculation based on received results is to be performed it will be desired that the calculation runs once every n second. In order to achieve this one must simply call those specific functions from inside the following if statement.

```
    /* This part of loop is synchronised
                            with system hardware timer */
    if(systemTimer >= TICK_RATE){
      systemTimer = 0;
    }
  }
}
```

7.1.2. Auxiliary Functions

Aside from the main function the Access Point program contains two other functions that are worth mentioning.

The first is the Timer B Interrupt Service Routine. Timer B is set to produce an interrupt each second and the function simply drives the system time.

```
#pragma vector=TIMERB0_VECTOR
__interrupt void TB0_ISR (void)
{
      systemTimeID++;
      systemTimer++;
}
```

Apart from the system time represented by the variable systemTimeID we also have a system timer. The timer can be used to produce periodic execution paths with 1 second precision. For example if we want to execute a certain function in the main program loop once every three seconds we simply must check if the timer >= 3 and execute the function then set the systemTimer variable to 0 again.

The second function to mention in the Access Point program is the SimpliciTi callback function.

```
static uint8_t sCB(linkID_t newLinkID)
{
	receiveLinkID = newLinkID;
	if(newLinkID){
		newFrame++;
	}
	else{
		newPeer++;
	}
	return 0;
}
```

This function runs under ISR context and is called by the Interrupt Service Routine belonging to the CC2500 interrupt pin connected to the MSP430F1611. The function is called every time a frame is received. The parameter newLinkID is either 0 if the peer that sent the frame is not yet linked with the Access Point, or non-zero in which case it represents the local link ID of one of the already linked peers. By saving this value into a variable one can identify exactly which of the connected peers sent the frame. Based on whether the peer is linked or not, the function activates either the new frame semaphore or the new peer semaphore which will trigger the respective program paths in the main loop. The function returns 0 which means the frame will be stored in the received frames queue for later processing (e.g. by using SMPL_Receive in the main loop). It is also possible to return 1 in which case the frame will simply be discarded. While not really useful for this example, this option can prove useful in a large network of fast sending End Devices to limit their access to the AP by ignoring some of their frames. It should also be noted that if the End Device has sent the frame with the auto-acknowledge option activated, if one wishes to discard the frame but still acknowledge that it was received one must send the acknowledge manually from this function before calling return 1. This way this case distinguishes itself from the case where on the End Device the frame receive was not acknowledged which would mean the send failed.

7.2. End Device Software Implementation

The End Device program is designed to allow easy porting to other platforms. This is why the connection and data transfer related code was separated from the main program. It should also be noted that the contents of the frames to be sent are easily modifiable but should be maintained in the present sequence.

7.2.1. Main Program Loop

The main program loop example calls the synchronizeWithAP() function and sets different contents for the buffer of the frame to be sent. It uses a timer to execute each loop every 3 seconds (this can be changed via a macro) and it puts the CPU to sleep otherwise. Note that once the CPU has entered low power mode, program execution will stop in that specific point and will be resumed after an interrupt (the only code that will still execute under sleep) has cleared the low power mode bits.

Please note that the following section contains code that has been broken down into snippets to allow text explanations in between. It should however be considered as a whole as implied by the different indentation of each line.

As the main function begins, the Board Support Package, serial interface and system timer are initialized. The statusFlags variable, local to the main function, will be used to store the different flags provided by the synchronizeWitAP function that handles the network communication. It is desired to know the status of the connection in the main function in order to implement dependencies to other type of code such as calculations without having to access variables that belong to the SimpliciTi part of the code directly (library-oriented style).

```
void main (void)
{
  uint8_t statusFlags = 0;    /* Local flag byte for the
                                 synchronise function */
  BSP_Init();
  COM_Init();     /* Initialise serial communication */
  START_TIMER();
  while(1){
    if(systemTimer >= TICK_RATE){   /* Run main loop each
                                       TICK_RATE seconds */
    /* Reset main timer */
    systemTimer = 0;
    /* The timeID of the next frame to be sent is our
                                         system's timeID */
    frameTimeID = systemTimeID;
    /* Prepare send buffer for next transfer
                                  (frame type and time ID) */
    sendBuffer[0] = 1;
    sendBuffer[1] = (uint8_t)frameTimeID;
    sendBuffer[2] = (uint8_t)(frameTimeID>>8);
    sendBuffer[3] = (uint8_t)(frameTimeID>>16);
    sendBuffer[4] = (uint8_t)(frameTimeID>>24);
    /* Count attempts of the next transfer */
    receiveAttempt++;
    sendAttempt++;
```

After the initialization, the while(1) loop begins and the main part of its code will be executed based on the system timer (once every 3 seconds in this case). The system timer is set to 0 again and the contents of the buffer of the frame to be sent (a synchronization frame in this case) are filled with the End Device’s system time in an encoded form. The variables receiveAttempt and sendAttempt are increased prior to the next part where a send attempt and a receive attempt will take place. These variables are used to form statistics such as send and receive success ratio which are used in testing the reliability of the connection.

The synchronization frame is sent and a frame is received by calling the synchronizeWithAP function with the SEND_RECEIVE switch. Based on the different flags returned by this function (encoded bitwise in a single byte) the receiveSuccess and sendSuccess variables are incremented in case of successful send and receive. These are used in order to form statistics on the connection’s functionality coupled with the statistics variables from before (e.g. receiveSuccess / receiveAttempt x 100 gives the success rate of receiving a frame in percentages).

```
/* Run main synchronisation function and
                                  save returned flag byte */
/* The sent frame is ment to let the AP know we are awake.
In the received frame we expect a command and the AP time */
statusFlags=synchroniseWithAP(sendBuffer,
                              sizeof(sendBuffer),
                              receiveBuffer,
                              &receiveBufferlength,
                              SEND_RECEIVE);
/* Check flags and count successful receive and/or send */
if(GET_FLAG(FRAME_RECEIVED,statusFlags) == TRUE){
  receiveSuccess++;
}
if(GET_FLAG(FRAME_SENT,statusFlags) == TRUE){
  sendSuccess++;
}
/* Output serial text information based on status flags */
outputStatus(statusFlags);
/* Build frame time ID from last received frame */
frameTimeID = ((uint32_t)receiveBuffer[1])+
              (((uint32_t)receiveBuffer[2])<<8)+
              (((uint32_t)receiveBuffer[3])<<16)+
              (((uint32_t)receiveBuffer[4])<<24);
/* The time ID received from the Access Point
                            becomes the new system time */
systemTimeID = frameTimeID;
/* If frame exchange was successful send command
                  result in a send only synchronisation */
```

The function outputStatus was implemented for debugging purposes and sends the current status of the connection together with the accumulated statistics to the serial interface so that it may be displayed on a PC connected to the End Device via USB.

The time received in the command frame is decoded and stored as the ED’s system time. This is not done in a single step but by making use of the intermediary variable frameTimeID to allow eventual time comparison policies such as duplicate frame detection.

If the previous synchronization succeeded the send buffer will be loaded with a result frame (for this example the received time from the previous step). The sendAttempt variable is increased prior another synchronization attempt, this time with the switch SEND_ONLY as the ED does not expect a reply in this case and if the send was successful, based on flag check, the sendSuccess variable will also be incremented.

```
      /* If frame exchange was successful send command result in
                                a send only synchronisation */
      if(GET_FLAG(FRAME_RECEIVED,statusFlags) == TRUE){
        /* Prepare send buffer for next transfer
                                   (frame type and time ID) */
        sendBuffer[0] = 2;
        sendBuffer[1] = (uint8_t)frameTimeID;
        sendBuffer[2] = (uint8_t)(frameTimeID>>8);
        sendBuffer[3] = (uint8_t)(frameTimeID>>16);
        sendBuffer[4] = (uint8_t)(frameTimeID>>24);
        /* Count send attempt */
        sendAttempt++;
        /* The sent frame should be the result to
                            the command previously received */
        statusFlags=synchroniseWithAP(sendBuffer,
                                      sizeof(sendBuffer),
                                      receiveBuffer,
                                      &receiveBufferlength,
                                      SEND_ONLY);
        /* Check flag for successful send */
        if(GET_FLAG(FRAME_SENT,statusFlags) == TRUE){
          sendSuccess++;
        }
        /* Output serial text information based on status flags */
        outputStatus(statusFlags);
      }
```

The status of the last communication attempt is displayed as text using the serial interface by calling the outputStatus again.

Following the if part of the first if statement (that checks the system timer) the else part will put the system to sleep.

```
      else{
        /* If timer interval not met go to sleep. Will
                             wake up on next timer interrupt */
        SLEEP();
      }
    }
}
```

This makes sure that the previously described code section runs each 3 seconds and should it complete execution before the 3 seconds are up the whole system goes into low power mode and will be woken up on the next 3 second interval by the timer interrupt.

7.2.2. RF Synchronization Function

The synchronizeWithAP function is responsible for creating and maintaining a data transfer connection with the Access Point. This is the main function that should be implemented in a way that makes porting to other platforms easier. It does not work with any of the global variables and the function body provides everything needed to implement the function (considering that the SimpliciTi API is already included, of course). The function has the ability to execute a 2 way transfer (first send and then receive) by using provided data buffers in its parameter list, and, by changing a simple switch, a 1 way transfer (send only). It fits, thus, the End Device implementation strategy of waking up from sleep, letting the Access Point know it is ready to receive commands via sending a frame, receiving a command frame, optionally executing command and sending the result back to the AP. Worth mentioning is that the function performs LED control to show the current network operation visually.

```
/*****************************************************************\
Function:             synchroniseWithAP

Description           Executes send/receive transfer with the AP
                      and also controls green/red (send/receive) LEDs

Input Parameters:     sendBuffer - pointer to the data to be sent
                      sendBufferlength - length of the send data
                                         array
                      receiveBuffer - pointer to the data array where
                                      received message will be stored
                      receiveBufferlength - pointer to the variable
                                            where the length of the
                                            received message will be
                                            stored
                      mode - switch to choose between send/receive or
                             send only (receive wait will be skipped)

Return:               1 byte containing status flags (each flag is a
                                                 set or cleared bit)
\*****************************************************************/

uint8_t synchroniseWithAP(uint8_t* sendBuffer,
                          uint8_t sendBufferlength,
                          uint8_t* receiveBuffer,
                          uint8_t* receiveBufferlength,
                          uint8_t mode)
```

The function employs several static variables. This is of great importance as locally declared static variables in a function will receive their initialization value when the space is allocated and will keep the value they contain between different function calls. That is to say that if a variable is set to 1 for example on 1 function run, the next time the function will be called that variable will still hold the value 1. This helps a great deal in keeping the status of the connection in between synchronizations.

```
#define TRUE 1
#define FALSE 0
#define CLEAR_FLAG(flag,byte) st(byte &= ~flag;)
#define SET_FLAG(flag,byte) st(byte |= flag;)
#define GET_FLAG(flag,byte) (((byte & flag) == flag)?TRUE:FALSE
#define JOINED 1
#define LINKED 2
#define FRAME_SENT 4
#define FRAME_RECEIVED 8
smplStatus_t SMPL_Status = SMPL_SUCCESS;
/* Static variables will maintain their value from one function
                                                call to the next */
static linkID_t APLinkID = 0;
static uint8_t statusByte = 0;
static uint8_t failedAttempts = 0;
```

Several helper macros are defined for this function in order to facilitate flag control from within the status byte (keep in mind that the flags are bitwise encoded).

The function will perform several tasks based on whether specific flags are set in the statusByte variable. The first task to perform is to check if the End Device has already joined a network and if not, to do so.

```
/* Switch on radio, starts in idle state */
SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_AWAKE, 0);
/* Check for joined flag and attempt join as needed */
if(GET_FLAG(JOINED,statusByte) == FALSE){
  uint8_t tryCount = MAX_RETRIES;
  LED_ALL_ON();
  /* Attempt join several times until successful or MAX_RETRIES */
  do{
    SMPL_Status = SMPL_Init(sCB);
    tryCount--;
  }while(tryCount && SMPL_Status != SMPL_SUCCESS);
  /* If successfuly joined set flag */
  if(SMPL_Status == SMPL_SUCCESS){
    SET_FLAG(JOINED,statusByte);
  }
}
```

After turning the radio on, the function will attempt to join a network a certain number of times (MAX_RETRIES) and if successful set the appropriate flag in the status byte. By not setting the flag on next function execution a new join attempt will be made.

If the End Device has already joined a network but is not yet linked a link attempt will take place. Again the function will try several times before giving up and if successful will set the link flag. However if the whole procedure is not successful more times in a row eventually the joined flag will be removed and the End Device will try to join a new network (or rejoin the old one).

```
/* If joined check for link flag and link with AP */
if((GET_FLAG(JOINED,statusByte) == TRUE) &&
   (GET_FLAG(LINKED,statusByte) == FALSE)){
  uint8_t tryCount = MAX_RETRIES;
  /* Attempt link several times until successful or MAX_RETRIES
                                with a delay between attempts */
  do{
    LED_ALL_ON();
    SMPL_Status = SMPL_Link(&APLinkID);
    tryCount--;
    NWK_DELAY(BLINK_DELAY);
    LED_ALL_OFF();
  }while(tryCount && SMPL_Status != SMPL_SUCCESS);
  /* If successfuly linked set flag and clear failed attempts*/
  if(SMPL_Status == SMPL_SUCCESS){
  SET_FLAG(LINKED,statusByte);
    failedAttempts = 0;
  }
  /* If all link attempts have failed clear the flag and count a
                                 complete fail of the function */
  else{
    CLEAR_FLAG(LINKED,statusByte);
    failedAttempts++;
    /* If function has failed several times AP is not online
                                  anymore so retry to join */
    if(failedAttempts > MAX_RETRIES){
      failedAttempts = 0;
      CLEAR_FLAG(JOINED,statusByte);
    }
  }
}
```

If the End Device has joined the network and is already linked to the AP, the function will attempt to send the frame provided by the send buffer received as input parameter. After attempting several times the FRAME_SENT flag will be set on successful send or cleared if the send failed.

```
/* If successfuly connected try to send */
if((GET_FLAG(JOINED,statusByte)&&
    GET_FLAG(LINKED,statusByte))==
    TRUE){
  uint8_t tryCount = MAX_RETRIES;
  do{
    LED_GREEN_ON();
```

```
      SMPL_Status = SMPL_SendOpt(APLinkID,
                                 sendBuffer,
                                 sendBufferlength,
                                 SMPL_TXOPTION_ACKREQ);
      tryCount--;
      NWK_DELAY(BLINK_DELAY);
      LED_GREEN_OFF();
    }while(tryCount && SMPL_Status != SMPL_SUCCESS);
    /* On successful send set the flag and clear failed attempts */
    if(SMPL_Status == SMPL_SUCCESS){
      SET_FLAG(FRAME_SENT,statusByte);
      failedAttempts = 0;
    }
    /*If send has failed clear the flag and count a complete fail*/
    else{
      CLEAR_FLAG(FRAME_SENT,statusByte);
      failedAttempts++;
      /* If the function has failed to send several times in a row
                                AP is down so retry to link */
      if(failedAttempts > MAX_RETRIES){
        failedAttempts = 0;
        /* If AP doesnt respond to ping also try to rejoin */
        if(SMPL_Ping(APLinkID) == SMPL_SUCCESS){
          CLEAR_FLAG(JOINED,statusByte);
        }
        CLEAR_FLAG(LINKED,statusByte);
        SMPL_Unlink(APLinkID);
        APLinkID = 0;
      }
    }
  }
```

If the send procedure fails multiple times in a row the End Device will try to see if the Access Point is still online using the SMPL_Ping function and if that fails as well the function will unset the JOINED and LINKED flags. On the next function run the End Device will try to re-join and re-link.

If the SEND_RECEIVE switch was used and the last send procedure was successful the function will also attempt to receive a reply from the Access Point (waiting for a frame in case the send failed makes no sense as nothing will be received). First the radio will be switched to receive mode and then the End Device will enter low power mode. This is essential as it allows just the right time to be spent waiting (because the exact duration varies and cannot be known beforehand). After the End Device has entered low power mode it can only be woken up by interrupts that explicitly clear low power mode status. In the case of this program there are 2 such functions. One is called on each received frame by the SimpliciTi underlying protocols and the other is from the system timer module. The exact mechanics of these two will be explained in the next section. For now it is only important to know that the End Device will wake up either on the next received frame or latest after at least one second of sleep.

```
  /* If frame has been sent successfuly (implies fully connected as
well) try to receive */
  if(GET_FLAG(FRAME_SENT,statusByte) == TRUE && mode ==
SEND_RECEIVE){
    /* Turn on receive mode */
    SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_RXON, 0);
    /* Enter low power mode, once the message is received an
interrupt will wake up the CPU */
    SLEEP();
    SMPL_Status = SMPL_Receive(APLinkID, receiveBuffer,
receiveBufferlength);
    if(SMPL_Status == SMPL_SUCCESS){
      SET_FLAG(FRAME_RECEIVED,statusByte);
    }
    else{
      CLEAR_FLAG(FRAME_RECEIVED,statusByte);
    }
  }
  else{
    CLEAR_FLAG(FRAME_RECEIVED,statusByte);
  }
  /* All work done, put radio to sleep */
  SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_SLEEP, 0);
  /* Return the status flags */
  return statusByte;
}
```

If the frame is successfully received the FRAME_RECEIVED flag is set, otherwise it is cleared. At the end of the function the radio is switched back to low power mode (note that the CPU remains active) and the status byte is given as return value.

7.2.3. Auxiliary Functions

Several auxiliary functions are used by the End Device program. Some of them are Interrupt Service Routines while others help with debugging.

Because the End Devices have to be energy consumption efficient they must frequently enter low power modes. For this purpose a macro has been defined in order to improve code readability.

```
/* Turn on Low Power mode */
/* Set a flag to indicate sleep mode to interrupt routines */
/* Enter low power mode */
#define SLEEP()                     \
st(                                       \
  systemSleep = 1;              \
  __bis_SR_register(LPM0_bits + GIE);\
)
```

The global variable systemSleep is set to 1 to prevent the timer interrupt service routine from waking up the End Device before a full second of sleep has passed. Because waiting for a frame to be received is done in sleep mode it may happen that the timer ISR is triggered before the receive frame ISR. By making the timer ISR wait one more second before waking up the system this problem is avoided and the correct time is spent waiting for a frame.

```
#pragma vector=WDT_VECTOR
__interrupt void WDT_ISR(void)
{
  systemTimeID++; /* Increment time counters */
  systemTimer++;
  if(systemSleep == 1){ /* Check sleep mode flag */
    systemSleep = 0;    /* Reset the flag but dont wake the
                        CPU. It will be woken up the next ISR run */
  }/* This guarantees at least a second of sleep without the need of
                                   shifting the 1 second interval */
  else{
    __bic_SR_register_on_exit(LPM3_bits); /* Wake up the CPU */
  }
}
```

The timer used on the GASB Sensor Platform is actually the watchdog timer set in interval mode because the MSP430F1611 that resides on the board only has two normal timer modules, one is used by SimpliciTi and one is used by the measuring software that will be implemented. The timer ISR guarantees at least a second of sleep by checking the global flag systemSleep.

```
/*******************************************************************\
Function:         sCB

Description:      SimpliciTi callback function. Runs in ISR context
                                           on each frame received

Input Parameters: newLinkID - the link ID from which the frame was
                              received (will be 0 if the frame comes
                              from a peer that tries to link the
                              first time)

Return:           0 if frame will be processed by application or 1 if
                                           frame should be discarded
\*******************************************************************/

static uint8_t sCB(linkID_t newLinkID)
{
      LED_RED_ON();
      NWK_DELAY(BLINK_DELAY);
      LED_RED_OFF();
      return 0;
}
```

The SimpliciTi callback function is much simpler than in the case of the Access Point. All it does is signal the frame that was received via LED blink and stores the received frame in the queue for later processing.

The outputStatus function, while not really essential for the functionality of the device, helps a great deal while debugging in order to check incoming and outgoing messages and send this information to a PC via serial interface.

```
/*****************************************************************\
Function:                outputStatus

Description:             Uses the flags byte to output text
                         information through the serial interface
                         concerning connection status (for debugging
                         purpouses)

Input Parameters:        statusFlags - byte containing connection
                                       flags

Return:                  nothing
\*****************************************************************/

void outputStatus(uint8_t statusFlags){
  /* First print the system time and receive and transfer success
rates */
  TXString(stringBuffer,
           sprintf(stringBuffer,
                   "\n\r [%u:%02u:%02u] ",
                   TID_HOURS(systemTimeID),
                   TID_MINUTES(systemTimeID),
                   TID_SECONDS(systemTimeID)));
  TXString(stringBuffer,
           sprintf(stringBuffer,
                 "[R:%5.1f%% T:%5.1f%%] ",
                 ((float)receiveSuccess)/((float)receiveAttempt)*100,
                 ((float)sendSuccess)/((float)sendAttempt)*100));
  /* Print relevant information for each connection state */
  switch(statusFlags){
    case 0:
      TXString(stringBuffer,
               sprintf(stringBuffer,"Joining network.. "));
      break;
    case JOINED:
      TXString(stringBuffer,
               sprintf(stringBuffer,"Linking to AP.. "));
      break;
    case JOINED+LINKED:
      TXString(stringBuffer,
               sprintf(stringBuffer,"Send frame failed "));
      break;
    case JOINED+LINKED+FRAME_SENT:
      frameTimeID=((uint32_t)sendBuffer[1])+
                  (((uint32_t)sendBuffer[2])<<8)+
                  (((uint32_t)sendBuffer[3])<<16)+
                  (((uint32_t)sendBuffer[4])<<24);
      TXString(stringBuffer,
               sprintf(stringBuffer,
                       "Sent frame: type=%u timeID=%u:%02u:%02u ",
                       sendBuffer[0],
                       TID_HOURS(frameTimeID),
                       TID_MINUTES(frameTimeID),
                       TID_SECONDS(frameTimeID)));
      break;
    case JOINED+LINKED+FRAME_SENT+FRAME_RECEIVED:
      TXString(stringBuffer,
               sprintf(stringBuffer,
                       "Sent frame: type=%u timeID=%u:%02u:%02u ",
                       sendBuffer[0],
                       TID_HOURS(frameTimeID),
                       TID_MINUTES(frameTimeID),
                       TID_SECONDS(frameTimeID)));
      TXString(stringBuffer,
               sprintf(stringBuffer,
                     "Received frame: type=%u timeID=%u:%02u:%02u ",
                       receiveBuffer[0],
                       TID_HOURS(frameTimeID),
                       TID_MINUTES(frameTimeID),
                       TID_SECONDS(frameTimeID)));
      break;
    default:
      break;
  }
}
```

The outputStatus function can easily be removed as it only uses global variables and the statusByte defined in the main program. While debugging one can connect both the End Device and the AccessPoint to a PC and use a terminal software that can display information from two separate COM ports in the same time to check the data traffic in real time.

7.3. Common Libraries

Since the Access Point needs to communicate with the PC via a Serial-to-USB interface, a library must be developed to handle this connection. It also makes sense to implement this library on the End Devices in order to send debugging information to the PC also via Serial-to-USB. The library is composed of two files: virtual_com_cmds.h and virtual_com_cmds.c. In order to add the library to a project one must simply include the library's header file where function prototypes are defined. The implementation of these functions is shown below.

```
void COM_Init(void)
{
      U1CTL        |=    SWRST;
      U1CTL        |=    CHAR;
      U1CTL        &=    ~(SYNC + SPB + PENA);
      U1TCTL |=    SSEL1;
      U1BR0        =     0x80;
      U1BR1        =     0x00;
      UMCTL1 =     0x00;
      ME2    |=    (URXE1 + UTXE1);
      U1CTL        &=    ~SWRST;
}
```

The COM_Init function initializes the UART of the microcontroller in SPI mode with a baud rate of 9600, no parity bits and one stop bit. Depending on the platform the library is used on this function needs to be modified in terms of register names and register value macros.

```
void TXString( char* string, int length )
{
  int pointer;
  for( pointer = 0; pointer < length; pointer++)
  {
    volatile int i;
    U1TXBUF = string[pointer];
    while (!(IFG2&UTXIFG1));            // USCI_A0 TX buffer ready?
  }
}
```

The TXString function receives a pointer to a string and its length as parameters and sends that to the UART TX buffer. It also waits for the ready flag to be set by the microcontroller. It can be used very easily in combination with sprintf() which prints formatted data to a string of choice and returns the number of successfully written characters.

```
TXString(stringBuffer,sprintf(stringBuffer,"A test string "));
```

As seen in the example above the first parameter is a char array and the second parameter is the string length. This is because the parameters are evaluated in C from last to first so sprintf() will be executed first and gets a chance to modify stringBuffer and return the number of correctly written characters which can be used as string length. Passing the length is necessary even though stringBuffer is actually a char array because passing arrays to functions causes arrays to decay into char pointers meaning they still contain information about where the string begins but the information about the size of the string is lost.

8. Conclusion and Outlook

At the time of writing this document, the main goal of the project has been completed and as such many doors have been opened in the further development of both the software framework concept and the hardware configurations and usages.

The hardware provided at the start of the project, namely the GASB Sensor Platform together with the GASB sensors and the ez430RF2500 target boards, has been configured according to the project plan and, together with the developed software, the sensor platforms are able to communicate with the computer using radio links and the received data from the Access Point is available for processing on the PC.

By combining the low power characteristics of the MSP controllers with those of the CC2500 RF modem the control and radio part of the platform is optimized with respect to power consumption. Considering that the GASB gas sensors also have low power consumption capabilities all there is left to do is to optimize the peripheral hardware on the devices. Testing the power consumption change only makes sense when the complete platform has been optimized.

The software framework that has been developed works as intended and has been thoroughly tested. It has proven to be quite stable, though it is hard to predict that that would not change after adding new user application content. However having a solid base to start on the next project should provide a big step in early development.

While the developed software may not seem to have a very precise form, one should consider that having flexibility as purpose has as end effect a structure with many undefined paths which in turn allow a great variation in usages. The framework developed in this project was born out of a great quantity of research performed on the used hardware and software protocols, and without this research most of this implementation would have been cumbersome and time-consuming. Given the current advancements in both hardware and software, basing new ideas on existing knowledge and technical information is not only recommended, but most of the time required, in order to shorten development time.

For the author the project has provided a great source of knowledge regarding wireless communication protocol concepts, hardware set-ups of remote communication devices and, in general, hardware platform development and software design.

In closing, the author would like to thank his supervisor, Dr Roland Pohle together with all the colleagues of the Siemens Corporate Technologies HW2 Department for their great support and aid in the project and for providing a friendly working environment.

9. Bibliography

[1] M. Fleischer, Advances in application potential of adsorptive-type solid state gas sensors. 2008 Meas. Sci. Technol. 19 042001.

[2] Ch. Wilbertz, H.-P. Frerichs, I. Freund, M. Lehmann, Suspended-Gate- and Lundstrom-FET integrated on a CMOS-chip, Sens. Actuators A 123-124 (2005) 2–6

[3] CO2 sensors based on workfunction readout using floating gate FET devices with polysiloxanes sensing layers by R. Pohle, A. Tawil, O. von Sicard, M. Fleischer, H.-P. Frerichs, Ch. Wilbertz, I. Freund

[4] Sensors and Actuators B 120 (2007) 669–672 – Fire detection with low power fet gas sensors by R. Pohle, E. Simona, R. Schneider, M. Fleischer, R. Sollacher, H. Gaoa, K. Müller, P. Jauch, M. Loepfe, H.-P. Frerichs, C. Willbertz.

[5] Texas Instruments Online Documentation http://focus.ti.com/general/docs/techdocs.tsp?siloId=1 SimpliciTi Overview (REV. B) datasheet, MSP430 1xxx and 2xxx family datasheets, CC2500 RF Module datasheet.

A1. Source Codes

Due to size and formatting issues the complete source code solution will be attached in digital format on the CD containing the digital form of this document.

A2. Figure Index

A3. Definitions

Below is a list of important terms used throughout this document and their respective definitions:

- AP – Access Point
- ED – End Device
- BSP – Board Support Package
- Frame – A message sent with the SimpliciTi protocol
- API – Application programming interface
- MSP – Mixed Signal Processor
- ISR – Interrupt Service Routine