AN746

Si4012 Programming Guide

1. Introduction

This document gives an overview of how to configure and use the Si4012 radio transmitter and provides several simple software examples.

The Silicon Laboratories' Si4012 is a fully integrated, crystal-less CMOS high-data rate RF transmitter designed for the sub-GHz ISM band. This chip is optimized for battery powered applications requiring low standby current and high output transmit power.

The following operation examples are covered in the programming guide:

- How to configure the Si4012
- How to use the Si4012 transmitter for packet transmission in FIFO mode
- How to measure battery voltage

The latest example source codes are available on the Silicon Labs web site: www.silabs.com.

2. Hardware Platform

The source codes are provided for the LCD Base Board hardware platform that is part of the 4012-LCDK1W-434 and the 4012-LCDK1W-915 Si4012 EZRadio One Way Link Development Kits.The LCD Base Board platform is a demo and development platform for the EZRadio family of RF chips. It consists of an LCD Base Board and interchangeable RF Pico Boards.

2.1. The LCD Base Board

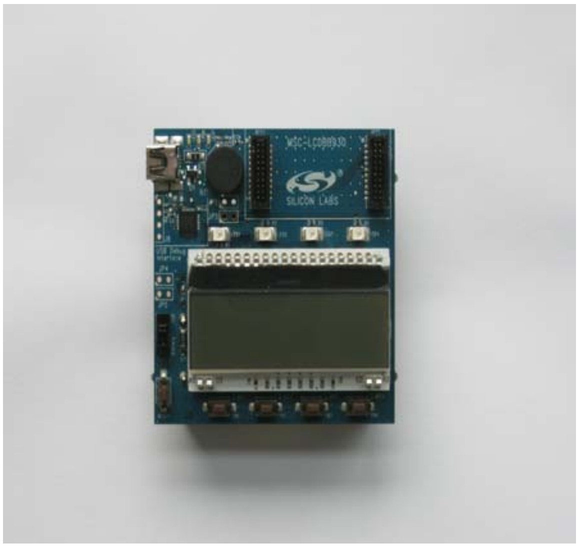


Figure 1. LCD Base Board

The board contains an LCD, four pushbuttons, four LEDs and a buzzer, connected to a Silicon Labs 8051F930 MCU (U2, under the LCD). The MCU is also connected to an RF Pico Board connector pair (RFP1, RFP2). The connection between the MCU and the EZRadio chip on the RF Pico Board is compatible with the RFStick. See Table 1 for details. A Silicon Labs USB to C2 debug interface (U4) is also integrated on the board, so the board can directly be connected via USB to the PC for downloading and debugging code on the U2 MCU.

Table 1. LCD Base Board Selection

Part Number	Board Description
MSC-LCDBB930-PER	LCD base board factory loaded with PER demo fw
MSC-LCDBB930-AES	LCD base board factory loaded with AES demo fw

2.2. RF Pico Board

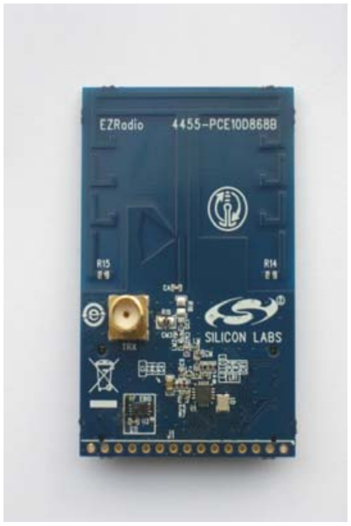


Figure 2. RF Pico Board

The RF Pico Board is a radio module that contains an EZRadio IC, matching network and pcb antenna. The RF output is selectable between the PCB antenna and a 50 Ω SMA output connector via a 0 Ω resistor. The boards also have a factory loaded board identification memory (EBID) that contains data that describes the board properties. Via the unified RF pico connector pair on the bottom side of the board, any RF pico board can be connected to the LCD base board.

The example code runs on the C8051F930 MCU of the MSC-LCDBB930 LCD base board and controls the Si4012 on the RF Pico Board using the SMBus interface. The signals listed in Table 2 are connected:

Table 2. Pin Assignments

Si4012			C8051F930	
Pin #	**Pin Name**	**Pin Function**	**Pin #**	**Pin name**
2	GND	Ground		Ground
5	VDD	Supply input		VDD
7	NIRQ	Interrupt status output, active low	20	P1.4
8	SDN	Shutdown input, active high	19	P1.5
9	SCL	SMBus clock input/output	25	P0.7
10	SDA	SMBus data input/output	26	P0.6

2.3. Setting up and Connecting the LCD Base Board to a PC

Power source of the platform can be selected with the power-supply selector switch (SW1) on the base board. If SW1 is in USB position, supply voltage is provided by the PC that is connected to the J7 mini USB connector. If SW1 is in Battery position, the supply voltage is provided by three AA batteries in the battery holder on the bottom side of the board. Current consumption of the RF part of the connected RF Pico Board (RFVDD) can be measured on JP3. Since JP3 is shorted by a PCB track on the bottom side of the board, the user have to cut the track if this feature is used.

Steps of connecting the platform to a PC:

- Connect an RF Pico Board to the LCD base board through RFP1 and RFP2.
- Select the desired power source with SW1 power selector switch.
- Connect the LCD base board to an USB port of the PC.
- Wait for Windows to install the driver of the debug interface if necessary.

3. Software Tools

There are two software tools provided by Silicon Labs to help EZRadio software development, the Wireless Development Suite (WDS) and the Silicon Labs Integrated Development Environment (IDE), both available on silabs.com.

3.1. Wireless Development Suite (WDS)

The recommended starting point for Si4012 development is the WDS. It can be downloaded from silabs.com and can be installed on a PC. After connecting one of the hardware platforms described in this document to the PC, WDS is able to identify the connected board by reading the EBID memories of the board.

The EZConfig Setup GUI is part of the Wireless Development Suite (WDS) program. This setup interface provides an easy path to quickly selecting and loading the desired configuration for the Si4012 device. The configuration table provides a list of preloaded, common configurations validated by Silicon Labs. EZConfig also allows for custom configuration to be loaded using the Radio Configuration Application. After the desired configuration is selected, the EZConfig setup automatically creates the configuration data that can be used to configure the EZRadio chip. The program then gives the option to configure directly the EZRadio chip of the connected hardware, to modify a selected demo code with the configuration and download it to the connected hardware, or to launch Silicon Labs IDE with the new configuration data preloaded into the user program. For more complete information on WDS and EZConfig usage, refer to the WDS User’s Guide, available on silabs.com.

3.2. Silicon Labs IDE

The Silicon Laboratories Integrated Development Environment (IDE) is a standard tool for program development for any Silicon Labs 8-bit MCUs including the C8051F930 that is used on the hardware platforms described in this document. The Silicon Laboratories IDE integrates a project manager, a source-code editor, source-level debugger, and an in-system flash programmer. The IDE interfaces to third party development tool chains to provide system designers a complete embedded software development environment. The Keil Demonstration Toolset includes a compiler, linker, and assembler and easily integrates into the IDE.

3.2.1. Downloading and Running the Example Codes

1. Connect the hardware platform to the PC according to the description of the used platform.
2. Start Silicon Labs IDE (IDE 4.40 or higher required) on your computer.
3. Select **Project→Open Project...** to open a previously saved project.
4. Before connecting to the target device, several connection options may need to be set. Open the **Connection Options** window by selecting **Options→Connection Options...** in the IDE menu.
5. Select USB Debug Adapter in the “Serial Adapter” section.
6. If more than one adapter is connected, choose the appropriate serial number from the drop-down list.
7. Check the “Power target after disconnect” if the target board is currently being powered by the USB Debug Adapter. The board will remain powered after a software disconnect by the IDE.
8. Next, the correct “Debug Interface” must be selected. Check the C2 Debug Interface.
9. Once all the selections are made, click the **OK** button to close the window.
10. Click the **Connect** button in the toolbar or select **Debug→Connect** from the menu to connect to the C8051F930 MCU of the platform.
11. Erase the flash of the C8051F930 MCU in the Debug→Download object code→ Erase all code space menu item.
12. Download the desired example HEX file either by hitting the Download code (Alt+D) toolbar button or from the Debug →Download object code menu item.
13. Hit the Disconnect toolbar button or invoke the Debug →Disconnect menu item to release the device from halt and to let it run.

4. Controlling the Si4012

The Si4012 has four pins to interface with the host MCU:

- SDN is the shutdown input of the IC. When it is pulled high or left open, the Si4012 goes to its shutdown state; when pulled low, the Si4012 wakes up.
- NIRQ is the interrupt request output that goes low when an enabled interrupt is triggered in the Si4012.
- SCL is the SMBus clock.
- SDA is the SMBus data.

4.1. Control

The SMBus interface is implemented as a bidirectional 2-wire interface (SCL, SDA) with the host configured as master and the Si4012 as slave. Both standard (100 kbs) and fast (400 kbs) modes are supported with 7-bit addressing. The device address is 1110000x, where x is the R/W bit. Since shutdown can be activated, wakeup can be initiated and interrupts can be polled using the SMBus interface. The minimal necessary control interface for the Si4012 is only two pins, SDA and SCL.

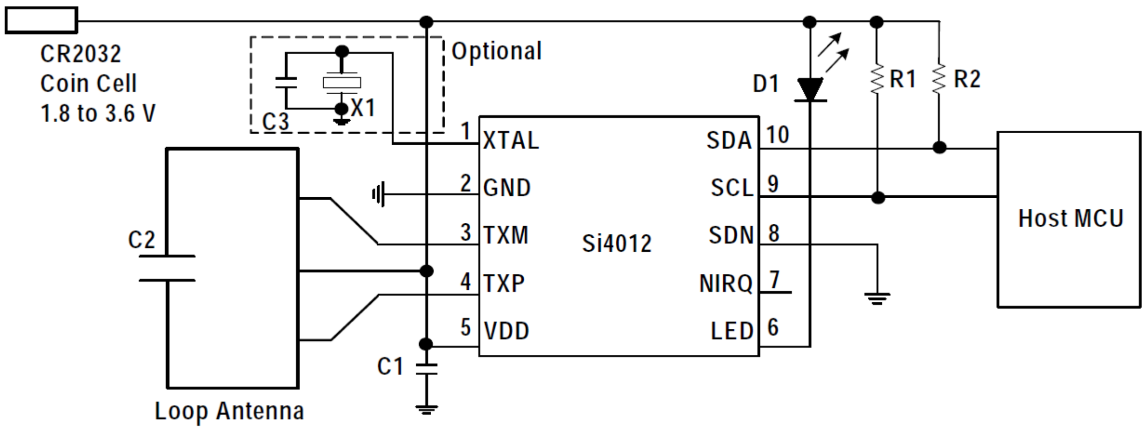


Figure 3. Minimal MCU Interface

All host commands consist of a 1-byte opcode followed by 0 or more arguments. All responses from the 4012 consist of a 1-byte top level status followed by 0 or more data values. Refer to the data sheet for details on the command structure and command details of the Si4012.

4.2. Initialization

After power on or wake up from shut down, the chip goes through an internal reset process and gets in an initial state.

Initialization of the chip can be done by setting some basic parameters of the chip operation according to the user's requirements. For calculating the parameters, a calculation spreadsheet, "si4012_calc_regs.xls", is provided by Silicon Labs. For details, see "AN564: Si4012 Calculator Spreadsheet Usage", available at www.silabs.com. The user inputs the required values into the spreadsheet:

- Center frequency
- Modulation type
- FSK deviation
- Total PA power
- Bit rate

and gets the PROPERTY values, such as:

- PA_CONFIG
- TX_FREQ
- MODULATION_FSKDEV
- BITRATE_CONFIG

in hex form that have to be used as parameters of the SET_PROPERTY command to be sent to the Si4012.

The commands typically used for initialization are:

- SET_PROPERTY
- SET_INT
- CHANGE_STATE
- LED_CTRL

To ease the use of these commands, a C header file called *Setup_Si4012.h* can be generated via the Wireless Development Suite.

The generated header file contains the precompiled commands according to the parameter values set on the Control Panel of WDS. Here is a part of the header file used in the examples:

```
#define Si4012_MODE_SELECTION 0x60, 0x00, 0x00
/*
       Selected mode: Standby
*/

#define Si4012_TUNE_INTERVAL 0x11, 0x21, 0x00, 0x0A
/*
       Tune interval: 10
*/

#define Si4012_MODULATION_FSKDEV 0x11, 0x20, 0x01, 0x18
/*
       Modulation type: FSK
       biFskDev: 24
*/

#define Si4012_CHIP_CONFIG 0x11, 0x10, 0x00
/*
       FSK deviation polarity:
       LDB First: OFF
       Use external Cristal: OFF
*/

#define Si4012_TX_FREQ 0x11, 0x40, 0x33, 0xC1, 0xE4, 0x58
/*
       Transmission frequency [MHz]: 868,344920
*/

#define Si4012_BITRATE_CONFIG 0x11, 0x31, 0x00, 0x60, 0x02
/*
       Data Rate [kbps]: 9,6
       Ramp Rate [us]:
*/
```

It is also possible to manually modify this file so that it includes multiple commands for alternative values of the same parameter.

To use the header file above, constant arrays have to be initialized with the defined commands first.

Here is the corresponding part of the example program:

```
//arrays initialized with precompiled commands
SEGMENT_VARIABLE(C_CHIP_CONFIG[], U8, SEG_CODE) = {Si4012_CHIP_CONFIG};
SEGMENT_VARIABLE(C_LED_INTENSITY[], U8, SEG_CODE) = {Si4012_LED_INTENSITY};
SEGMENT_VARIABLE(C_MODULATION_FSKDEV[], U8, SEG_CODE) = {Si4012_MODULATION_FSKDEV};
SEGMENT_VARIABLE(C_TUNE_INTERVAL[], U8, SEG_CODE) = {Si4012_TUNE_INTERVAL};
SEGMENT_VARIABLE(C_FIFO_THRESHOLD[], U8, SEG_CODE) = {Si4012_FIFO_THRESHOLD};
SEGMENT_VARIABLE(C_BITRATE_CONFIG[], U8, SEG_CODE) = {Si4012_BITRATE_CONFIG};
SEGMENT_VARIABLE(C_TX_FREQ[], U8, SEG_CODE) = {Si4012_TX_FREQ};
SEGMENT_VARIABLE(C_LBD_CONFIG[], U8, SEG_CODE) = {Si4012_LBD_CONFIG};
SEGMENT_VARIABLE(C_XO_CONFIG[], U8, SEG_CODE) = {Si4012_XO_CONFIG};
SEGMENT_VARIABLE(C_PA_CONFIG[], U8, SEG_CODE) = {Si4012_PA_CONFIG};
```

Then the SetProperty function is called that outputs the constant arrays (i.e., the commands) on the SMBus:

```
//-----------------------------------------------------------------
//   set up the radio using the precompiled commands in Setup_Si4012.h
//   (Setup_Si4012.h can be generated from WDS)
//----------------------------------------------------------------
       Si4012_Status = SetProperty(C_LED_INTENSITY, 3);
```

```
        Si4012_Status = SetProperty(C_CHIP_CONFIG, 3);
        Si4012_Status = SetProperty(C_MODULATION_FSKDEV, 4);
//      Si4012_Status = SetProperty(C_TUNE_INTERVAL, 4);
//      Si4012_Status = SetProperty(C_FIFO_THRESHOLD, 5);
//      Si4012_Status = SetProperty(C_BITRATE_CONFIG, 5);
        Si4012_Status = SetProperty(C_TX_FREQ, 6);
//      Si4012_Status = SetProperty(C_LBD_CONFIG, 6);
//      Si4012_Status = SetProperty(C_XO_CONFIG, 7);
//      Si4012_Status = SetProperty(C_PA_CONFIG, 8);
//-----------------------------------------------------------------------

U8 SetProperty(U8 * command, U8 clenght)
{

        SmbusStatus = SMBusWrite(SI4012_SLAVE_ADDRESS, clenght, command);
        SmbusStatus = SMBusRead(SI4012_SLAVE_ADDRESS, 1, &error);
        return error;
}
```

Because properties are initialized to a default value by the reset process after a power on or wakeup from shutdown, the user has to initialize a property only if its desired value is different from its default value. For this reason, some SetProperty calls are commented out in the code above to save time and battery power.

4.3. Interrupt Handling

The NIRQ pin of the Si4012 is used to issue interrupts to the host. The host should poll the NIRQ pin or use interrupts (NIRQ) to get interrupt status. The host can then read the interrupt status of the radio with GET_INT_STATUS command that also clears the interrupts at the same time. The GET_INT_STATUS command is also useful for polling the interrupt status of the Si4012 via the SMBus interface when the NIRQ pin is not connected.

4.4. How to Send a Packet Longer than 255 Bytes

The Si4012 device has a 255-byte transmit FIFO; however, it is possible to transmit longer packets than 255 bytes.

The FIFO in the Si4012 has two programmable thresholds and the chip can provide interrupts when the data in the transmit FIFO reaches these thresholds. The first threshold is the FIFO Almost Full Threshold. If the number of bytes filled into the FIFO reaches this level, then the radio can provide an interrupt for the MCU to start the packet transmission. The second threshold is the FIFO Almost Empty Threshold. If the number of bytes in the transmit FIFO reaches this level, then the radio can provide an interrupt for the MCU. Then the microcontroller will need to fill more data into the transmit FIFO to avoid FIFO underflow.

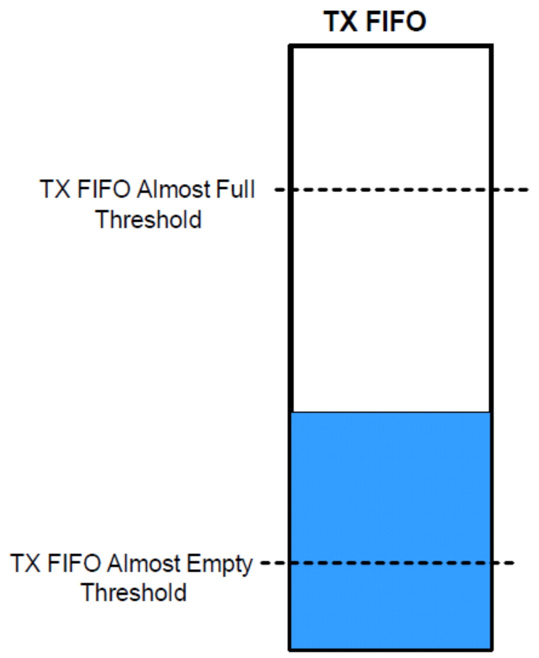


Figure 4. Transmit FIFO Status Signals

4.5. How to Measure the Battery Voltage

The Si4012 is able to measure its own supply voltage in two modes.

One mode of operation is to measure the supply voltage whenever the GET_BAT_STATUS command is invoked. The supply voltage is measured after the main current-consuming blocks of the device are temporarily turned on. These parts are turned off after the measurement is done. The measurement itself takes place after (LoadWaitTime x 17 µs) of wait time after the GET_BAT_STATUS command is invoked. If the LoadWaitTime parameter of the command is set to 0, the battery voltage is measured immediately without any load applied. The command gives back the result in mV.

In the other mode used for battery measurement, the Si4012 measures the supply voltage periodically with no load applied. If the battery voltage is below the LBD threshold, the Low Battery Detector Interrupt is raised. This periodical measurement happens only if Si4012 is in sensor mode and low battery interrupt is enabled. The period and threshold can be set in the LBD_CONFIG property.

4.6. Crystal Oscillator Tuning Capacitor

When crystal is used, the accuracy of the radio center frequency is determined by the parameters of the crystal (such as load capacitance, crystal accuracy, etc.) and the parasitic of the PCB associated with the crystal circuit. To reduce the impact of these crystal parameters and to guarantee safe startup of the oscillator, the user has to do the following:

- Check the crystal data sheet for the “Cload” capacitor value that should be placed across the crystal’s terminals to oscillate at the correct frequency
- If Cload > 14 pF, XoLowCap field of the XO_CONFIG property have to be set to 0. In this case, the input capacitance of the XTAL pin of the Si4012 is 5 pF, so a (Cload – 5)pF capacitor should be placed externally across the crystal terminals.
- If Cload < 14 pF, XoLowCap field of the XO_CONFIG property have to be set to 1. In this case, the input capacitance of the XTAL pin of the Si4012 is 2.5 pF, so the external capacitor placed across the crystal has to be (Cload – 2.5) pF.

The example program does not use a crystal.

5. Sample Projects

Several complete sample projects are provided in the AN746SW.zip file available at www.silabs.com.

5.1. General Project Structure

All the sample projects have a unified structure and common driver set. This chapter gives the reader a brief introduction of the structure of the sample projects.

5.1.1. Prerequisites

The settings in the sample project files assume that some Silicon Labs or third party software tools are already installed on the PC where the sample project going to be compiled. The tools need to be installed depends on the functionality to be used. The following list contains a complete set of such programs:

- **Silicon Laboratories IDE—**Used to open the preconfigured project files and manage the build process.
- **Keil C51 v9.0+** or **SDCC v3.0+—**Compilers to use in accordance with the Silicon Laboratories IDE to manage build process.
- **Silicon Labs Flash Programming Utility** *(optional)*—Needed only if programming outside the Silicon Labs IDE is necessary.
- **Make** *(optional)*—This tool is needed in case other compiler is used, or the build process takes place outside of the SiLabs IDE.

5.1.2. Supported Compilers

The projects come with two SiLabs IDE project file. The one that name ends with "_Keil" contains the settings for and is appropriate to compile with Keil's C51 tool chain. The other that ends with "_SDCC" is prepared to use the SDCC compiler tool chain. Please note, if you use the 2k limited evaluation license for Keil's C51 compiler, you may need to unlock it to 4k, as some project's size exceeds the 2k code size limit.

However, the sample projects can be complied not only with the two mentioned compiler, but with almost any ANSI C compiler for 8051 architecture with little or no modifications.

Each project already contains a "Makefile" in order to provide an easy and convenient way to compile the code outside the Silicon Labs' IDE with the toolchain of choice.

Each sample project described in this document contains a compiled version of the source code in Intel hex format that is widely supported by a variety of programming and debugging tools. The compiled file in the projects has been generated using the SiLabs IDE and the Keil C51 tool chain.

5.1.3. Directory Structure

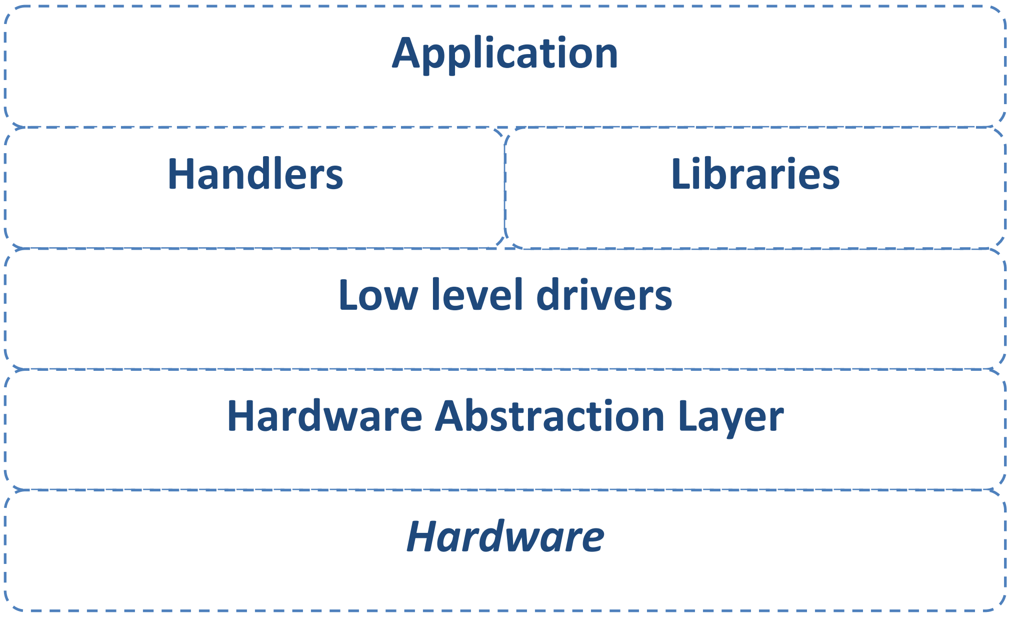
All sample project has a common directory structure, with separated source and project files in order to ease the understanding of the individual modules. For every sample project the following directories and files can be found in the main directory:

- **bin**—Contains the SiLabs project files for Keil and SDCC compilers and the Makefile if the make tool is used instead.
- **doc**—Doxygen generated documentation based on comments inside the source files in html format.
- **out**—The outputs of the compilation process come to this folder. After successful compilation, this directory contains for example the hex file, the linker output and the OMF file.
- **src**
  - application
  - drivers

  Directories containing the source files.
- ***Doxyfile***—In this file, the Doxygen documentation generator settings can be found.
- ***Cleanup.bat***—Batch file used to delete all files generated during build process.

5.1.4. Software Modules

In the sample projects the layered software approach is followed. There's a distinct scope for each software module and such modules can communicate through each other's API functions. The software modules are separated and focused to cover one specific task. The following figure shows the software layers and its relations.



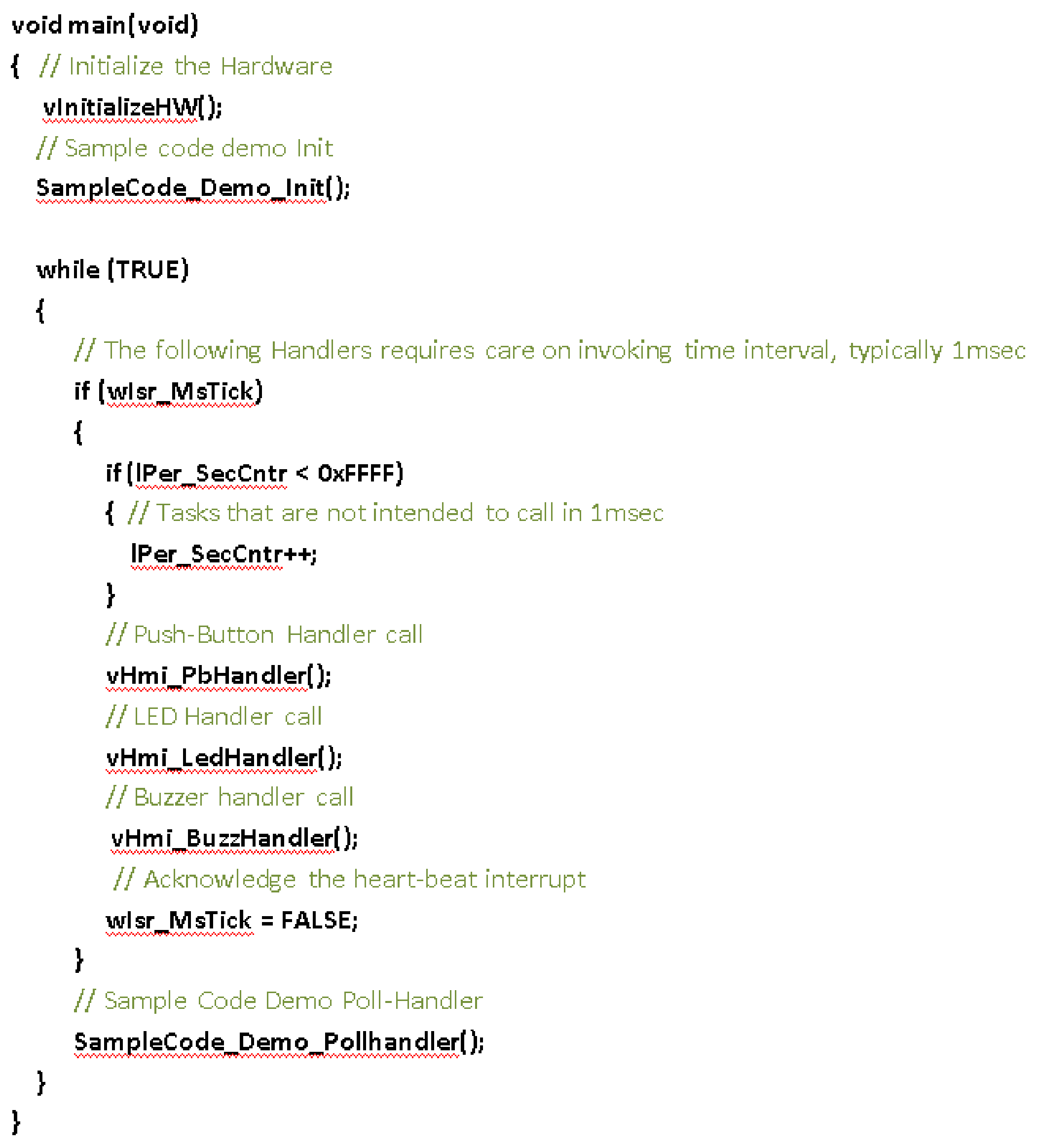
The individual software modules are separated into several source files. In the sample projects there is one header file that is included in the source files and collects the individual headers that need to be included.

Under the ***src*** folder the application related sources can be found in the ***application*** folder, the common modules (e.g., handlers, drivers) are located under the ***driver*** directory.

5.2. Sample Application Descriptions

5.2.1. Empty Project (Base_Project)

This sample project is intended to demonstrate the basic concept of setting up the Si4012 radio chip. This project is the essential core of the following three sample codes (CW_Transmit, PN9, TX). It is primarily responsible for initializing the MCU and RFIC as well. It also initializes the Human-Machine Interface (HMI) module related handlers, such as the push button handler, the LED handler, and the buzzer handler that are potentially useful for the rapid application development.



```
void main(void)
{  // Initialize the Hardware
    vInitializeHW();
   // Sample code demo Init
   SampleCode_Demo_Init();

   while (TRUE)
   {
       // The following Handlers requires care on invoking time interval, typically 1msec
       if (wIsr_MsTick)
       {
           if (lPer_SecCntr < 0xFFFF)
           {  // Tasks that are not intended to call in 1msec
              lPer_SecCntr++;
           }
           // Push-Button Handler call
           vHmi_PbHandler();
           // LED Handler call
           vHmi_LedHandler();
           // Buzzer handler call
            vHmi_BuzzHandler();
            // Acknowledge the heart-beat interrupt
           wIsr_MsTick = FALSE;
       }
       // Sample Code Demo Poll-Handler
       SampleCode_Demo_Pollhandler();
   }
}
```

In the main() function, the vInitializeHW() invokes initialize functions for the MCU IOs and peripherals. It also starts the HMI module handlers that are scheduled to run once in every 1 ms during the user application. These timing intervals are generated by the Timer2 peripheral overflow interrupt. Finally, it enables the MCU related internal interrupts.

```
void vInitializeHW(void)
{
 // Initialize the MCU peripherals
 vPlf_McuInit();

 // Start Timer2 peripheral  with overflow interrupt
 vTmr_StartTmr2(bTmr_Tmr2Both_c, wwTmr_Tmr2Periode.U16, TRUE,    bTmr_TxXCLK_00_c);

 // Initialize the push Button Handler
 vHmi_InitPbHandler();

 // Initialize the LED Handler
 vHmi_InitLedHandler();

 // Initialize the Buzzer Handler
 vHmi_InitBuzzer();

 // Enable configured interrupts
 mIsr_EnableAllIt();
}
```

In the main() function, the **SampleCode_Demo_Pollhandler()** is responsible for several tasks:

1. It checks if the Si4012 Pico board is connected to the LCD Baseboard.
2. Reads the RF EBID content in order to learn to which frequency band the Pico board is designed for. The possible options can be either 434 MHz or 915 MHz.
3. Configures the RFIC to the operating mode that is generated by the Wireless Development Studio.
4. Starts the user application program.

```
void SampleCode_Demo_Pollhandler()
{
 /* Sample_Code_ Demo state machine */
 switch (SampleCode_Demo_InternalData.SampleCode_Demo_State)
 {
     case SM_START:
         /* CHECK IF RF PICOBOARD CONNECTED TO THE LCD BASEBOARD OR NOT */

         /* Shutdown the Si4012 */
         Si4012_Shutdown();

         /* Check RF Pico board EBID availability */
         if (EBID_CheckAvailability() == TRUE)
         {    /* Picoboard is connected to the LCD baseboard */

             /* Read the content of the RF  EBID */
             if (EBID_SearchRadioRecord() != FALSE)
             {
                 /* Check if the RF Pico board is designed for either 434 MHz or 915MHz  */
                 if ( (EBIDInfo.EBID_Freqband & (1u << 9u)) || (EBIDInfo.EBID_Freqband & (1u << 4u)) )
                 {
                     /* Use the  settings which are already stored in the FLASH memory */
                     pSi4012_Config_Array = pSi4012_Config_Array_Default;
                 }
                 else
                 {  /*  Pico board is not designed to 434MHz or 915MHz  */
                     ERROR_HOOK;
                 }
                  SampleCode_Demo_InternalData.SampleCode_Demo_State = SM_STEP1;
                  break;
             }
         }
         /*  Waiting for Reset, because RF EBID is not available */
         ERROR_HOOK;
     break;
```

The **SampleCode_Demo_Pollhandler()** includes a state machine, and the different states represent the previously mentioned tasks.

1. SM_START:
- Turn off the Si4012 RFIC
- Check if the RF EBID is available on the Pico board
- Read the EBID content to determine the radio type and the frequency band

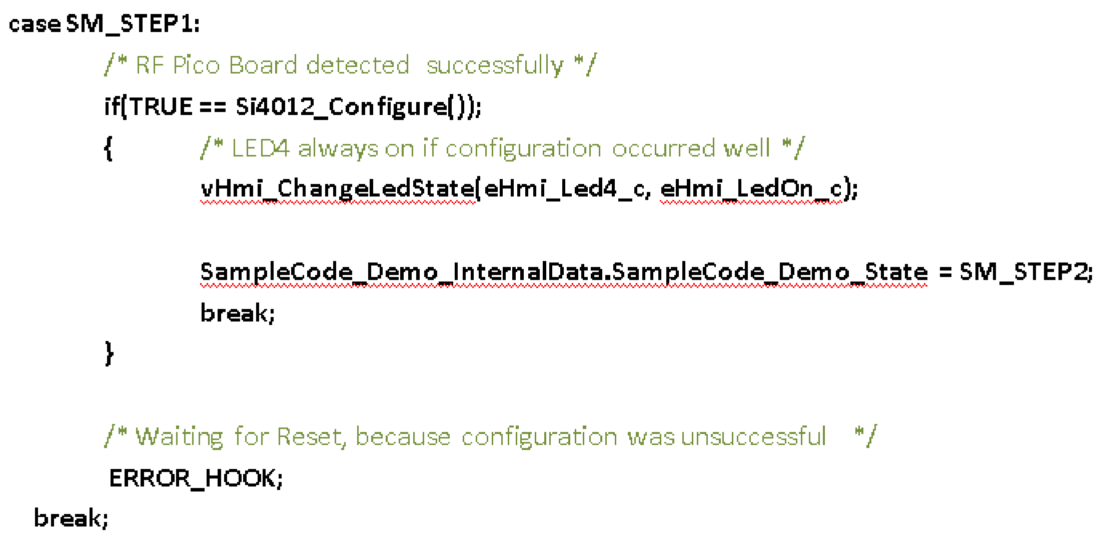
Having detected the RF Pico board successfully, the next state will configure the Si4012 RFIC.

2. SM_STEP1:
- Configure the radio with the WDS generated parameters
- Turn on the fourth LED on the LCD baseboard

Having configured the RFIC successfully, the next state will execute the user application code.

3. SM_STEP2:
- User application code can be started from this point.

The state of **SM_STEP1** simply configures the chip by calling the **Si4012_Configure()** function. The fourth LED will be turned on if there were no failures during the radio initialization. If any kinds of failures occurred either in SM_START or in SM_STEP1 states, for example, communication error, configuration error, or unavailable RF Pico board, the program execution will be forced to the **ERROR_HOOK** immediately. The fourth LED will then blink rapidly to indicate that the error detection occurred.

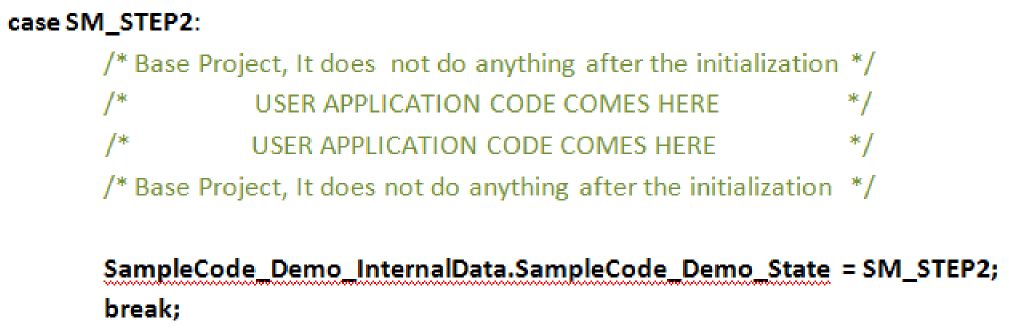


```
case SM_STEP1:
        /* RF Pico Board detected  successfully */
        if(TRUE == Si4012_Configure());
        {       /* LED4 always on if configuration occurred well */
                vHmi_ChangeLedState(eHmi_Led4_c, eHmi_LedOn_c);

                SampleCode_Demo_InternalData.SampleCode_Demo_State = SM_STEP2;
                break;
        }

        /* Waiting for Reset, because configuration was unsuccessful   */
         ERROR_HOOK;
  break;
```

The state of **SM_STEP2** can contain the user application code. Since it is an empty project, it does not execute any code after the radio initialization.



```
case SM_STEP2:
        /* Base Project, It does  not do anything after the initialization  */
        /*              USER APPLICATION CODE COMES HERE                   */
        /*              USER APPLICATION CODE COMES HERE                   */
        /* Base Project, It does not do anything after the initialization  */

        SampleCode_Demo_InternalData.SampleCode_Demo_State = SM_STEP2;
        break;
```

5.2.2. Continuous Wave Transmission (CW_Transmit)

Since the radio configuration happened according to the empty project, the only remaining task is to start the CW mode transmission with the Si4012_Start_CW_Mode() function. Having done the radio configuration, the transmission will start shortly thereafter.

```
case SM_STEP1:
        /* RF Pico Board detected  successfully */
        if(TRUE == Si4012_Configure());
        {
            /* Start transmission  in CW mode */
            Si4012_Start_CW_Mode();

            SampleCode_Demo_InternalData.SampleCode_Demo_State  = SM_STEP2;
            break;
        }

        /* Waiting for Reset, because  configuration was unsuccessful    */
        ERROR_HOOK;
        break;
```

To illustrate how to expand the code with dynamic behavior, **SM_STEP2** provides the user with a clear example of how to use the LCD baseboard push-buttons for turning in/out the CW transmission. After starting the CW mode, an LED on the RF Pico board will turn on, indicating that the mode is successfully activated. By pushing the second button, the user can stop the transmission with the **Si4012_Stop_CW_Mode()** function. By pushing the first button, the transmission can be restarted again.

```
case SM_STEP2:
        /* Check which pushbutton  was pushed*/
        gHmi_PbIsPushed(&boPbPushTrack, &woPbPushTime);

        if(boPbPushTrack & eHmi_Pb1_c)
        {     /*If button1  was pushed,  Restart transmission  in CW mode */
              Si4012_Start_CW_Mode();
        }

        if(boPbPushTrack & eHmi_Pb2_c)
        {
              /*If button2  was pushed,   Stop transmission  in CW mode */
              Si4012_Stop_CW_Mode();
        }

        SampleCode_Demo_InternalData.SampleCode_Demo_State  = SM_STEP2;
        break;
```

5.2.3. Pseudo Random Transmission (PN9)

Since the radio configuration happened according to the empty project, the only remaining task is to start the CW mode transmission with the Si4012_Start_PN9_Mode(U8 bModeSelector) function. Having done the radio configuration, the transmission will start shortly thereafter.

```
case SM_STEP1:
        /* RF Pico Board detected  successfully */
        if(TRUE == Si4012_Configure());
        {
              /* Start transmission  in PN9 0 mode */
              Si4012_Start_PN9_Mode(SI4012_PN9_0_MODE);

              SampleCode_Demo_InternalData.SampleCode_Demo_State  = SM_STEP2;
              break;
        }

        /* Waiting for Reset, because  configuration was unsuccessful   */
         ERROR_HOOK;
        break;
```

To illustrate how to expand the code with dynamic behavior, **SM_STEP2** provides the user with a clear example of how to use the LCD baseboard push-buttons for turning in/out the PN-9 transmission. After starting the PN-9 mode, an LED on the RF Pico board will turn on, indicating that the mode is successfully activated. By pushing the third button, the user can stop the transmission with the **Si4012_Stop_PN9_Mode()** function. By pushing the second and the third buttons, the transmission can be restarted again in PN-9 zero or in PN-9 one mode.

```
case SM_STEP2:
        /* Check which pushbutton  was pushed*/
        gHmi_PbIsPushed(&boPbPushTrack, &woPbPushTime);

         if(boPbPushTrack & eHmi_Pb1_c)
         {        /* If button1 was pushed , Stop transmission in PN9 mode */
                  Si4012_Stop_PN9_Mode();
                  /* Start transmission in PN-9 0 mode */
                  Si4012_Start_PN9_Mode(SI4012_PN9_0_MODE);
         }
         if(boPbPushTrack & eHmi_Pb2_c)
         {        /* If button2 was pushed,  Stop transmission in PN9 mode */
                  Si4012_Stop_PN9_Mode();

                  /* Start transmission in PN-9 1 mode */
                  Si4012_Start_PN9_Mode(SI4012_PN9_1_MODE);
         }

         if(boPbPushTrack & eHmi_Pb3_c)
         {         /* If button3 was  pushed,  Stop transmission in PN9 mode */
                  Si4012_Stop_PN9_Mode();
         }

         SampleCode_Demo_InternalData.SampleCode_Demo_State  = SM_STEP2;
         break;
```

5.2.4. Packet Transmission (TX)

Since the radio configuration happened according to the empty project, the only remaining task is to send packets with the Si4012_SendPacket(U8 bButtonNumber) function. Having done the radio configuration, SM_STEP2 waits for push button actions. Three different kinds of packets can be sent by pushing button1, button2, and button3. When a button is pushed, the appropriate LED on the LCD base board will blink once. The same packet will be sent over the air three times shortly after one another. After one packet has been sent successfully, the RF Pico board LED will blink once.

```
case SM_STEP2:
        /* Check which button was pushed*/
        gHmi_PbIsPushed(&boPbPushTrack, &woPbPushTime);

        if(boPbPushTrack & eHmi_Pb1_c)
        {       /* If Button1 pushed,  LCD baseboard->LED1 turns on */
                vHmi_ChangeLedState(eHmi_Led1_c, eHmi_LedBlinkOnce_c);

                /* Send packet 3 times */
                Si4012_SendPacket(eHmi_Pb1_c);
                Si4012_SendPacket(eHmi_Pb1_c);
                Si4012_SendPacket(eHmi_Pb1_c);
         }
        ...ETC...
        ...ETC...
        /* Push Button2 and Push Button3 actions are quite same */
        SampleCode_Demo_InternalData.SampleCode_Demo_State = SM_STEP2;
        break;
```

The following figure illustrates the packet structure used in this sample project.

PREAMBLE	SYNC WORD	CHIP ID	Button info	Rolling Counter	CRC
13 byte	2 byte	4 byte	1 byte	2 byte	2 byte

The CRC field is calculated over the CHIP ID, the button info, and the rolling counter to ensure error-free packet reception. The two-byte CRC is implemented in the CRC.h module by CRC_CalculateCRC16(U8* pbInputData, U8 bLengthOfInputData) function. The Si4012 related functions are implemented in the Si4012.h module. These functions can manipulate the radio such as turn off/on the radio chip, start/stop the different transmit modes (CW/PN9), send packets using the built-in FIFO, and set the internal properties via the radio's Application Programming Interface (API). The API offers the option to access all of the radio's internal properties. The host MCU is connected to the radio via SMBus. The SMBus related functions, which can provide access to the API, can be found in SMBus.h module under the drivers folder. The packet to be sent, including the preamble and the sync word pattern, are filled in the FIFO of the radio by the host MCU. The sample code related functions are listed below.

```
/* Basic radio ON/OFF commands */
 void Si4012_Shutdown(void);
 void Si4012_PowerUp(void);

/*API Configuration related functions */
 BIT Si4012_Configure(void);
 U8  Si4012_SetProperty(U8* pbCommand,U8 bLengthOfCommand);

 /* FIFO handles functions */
 U8 Si4012_InitFifo(void);
 U8 Si4012_FillFifo(U8 bButtonNumber);

 /* TX related functions */
 U8 Si4012_Start_Transmit(void);
 BIT Si4012_ClearInterrupt(void);
 U8 Si4012_Check_Transmit_Success();
 void Si4012_SendPacket(U8 bButtonNumber);

/*Pico board LED handle function*/
U8 Si4012_setLed(U8 bLedState);

/*Revision reader function*/
U8* Si4012_Get_Revision(void);

 /*CW mode related functions */
 U8 Si4012_Start_CW_Mode(void);
 U8 Si4012_Stop_CW_Mode(void);

 /* PN-9 related functions */
 U8 Si4012_Start_PN9_Mode(U8 bModeSelector);
 U8 Si4012_Stop_PN9_Mode(void);
```

The Setup_Si4012.h module configuration should be generated by the Wireless Development Studio, edited, and then added to the project in order to enable the Si4012_Configure() function to initialize the radio chip. During the configuration, the Si4012_SetProperty(…) function will initialize the API properties by sending the arrays of bytes beginning with prefix of SI4012_CONFIG_ARRAY_DATA_RADIO. The WDS generated properties are listed below.

```
#define SI4012_CONFIG_ARRAY_DATA_RADIO_MODE_SELECTION_DEFAULT          { 0x60, 0x00, 0x00 }
#define SI4012_CONFIG_ARRAY_DATA_RADIO_TUNE_INTERVAL_DEFAULT           { 0x11, 0x21, 0x00, 0x0A }
#define SI4012_CONFIG_ARRAY_DATA_RADIO_MODULATION_FSKDEV_DEFAULT       { 0x11, 0x20, 0x01, 0x5D }
#define SI4012_CONFIG_ARRAY_DATA_RADIO_CHIP_CONFIG_DEFAULT             { 0x11, 0x10, 0x00 }
#define SI4012_CONFIG_ARRAY_DATA_RADIO_TX_FREQ_DEFAULT             { 0x11, 0x40, 0x36, 0xAA, 0x24, 0x00 }
#define SI4012_CONFIG_ARRAY_DATA_RADIO_BITRATE_CONFIG_DEFAULT  { 0x11, 0x31, 0x00, 0x60, 0x02 }
#define SI4012_CONFIG_ARRAY_DATA_RADIO_XO_CONFIG_DEFAULT  { 0x11, 0x50, 0x00, 0x98, 0x96, 0x80, 0x00 }
#define SI4012_CONFIG_ARRAY_DATA_RADIO_PA_CONFIG_DEFAULT  { 0x11, 0x60, 0x01, 0x4D, 0x00, 0x80,0x7D,0xCC }
#define SI4012_CONFIG_ARRAY_DATA_RADIO_FIFO_THRESHOLD_DEFAULT   { 0x11, 0x30, 0xF0, 0x37, 0x20 }
#define SI4012_CONFIG_ARRAY_DATA_RADIO_LED_INTENSITY_DEFAULT    { 0x11, 0x11, 0x03 }
#define SI4012_CONFIG_ARRAY_DATA_RADIO_LED_CONTROL_DEFAULT      { 0x13, 0x00 }
#define SI4012_CONFIG_ARRAY_DATA_RADIO_FIFO_INTERRUPTS_DEFAULT { 0x63, 0x08 }
#define SI4012_CONFIG_ARRAY_DATA_RADIO_LBD_CONFIG_DEFAULT       { 0x11, 0x41, 0x09, 0xC4, 0x00, 0x3C }
```

The additional header files should be included in the common bsp.h located in the src/drivers directory. The bsp.h file already contains included headers:

```
/*        Application specific includes                    */
#include "compiler_defs.h"
#include "platform_defs.h"
#include "hardware_defs.h"

#include "..\application\application_defs.h"
#include "..\application\isr.h"

#include "..\drivers\smbus.h"
#include "..\drivers\control_IO.h"
#include "..\drivers\timer.h"
#include "..\drivers\pca.h"
#include "..\drivers\hmi.h"

#include "..\application\crc.h"
#include "..\application\ebid.h"
#include "..\application\Setup_Si4012.h"
#include "..\application\Si4012.h"
#include "..\application\SampleCodeDemo.h"

#endif //BSP_H
```

5.3. Common Software Modules

In the modules hierarchy the common software modules are located between the application and the hardware layers. It is a set of interfaces that provides possibilities to be able to control various peripherals on modular HW platforms. Registers can be initialized with pre-configured settings and peripherals can be enabled to start/stop their own processing. The major tasks of these software modules are to initialize the hardware elements and control its behaviors. The principle of their installation is to provide a façade for the upper layers. Functionally, the *User Application,* at the top of the hierarchy, can be independent of the hardware and its logical operation can remain unchanged even if the hardware has been modified later. It can be adapted to any device without encountering difficulties. All the modules in the following subsections except the human-machine interface module are mainly responsible for handling the dedicated internal peripherals such as the IO, the timers, the SPI and the PCA. The HMI holds them together so it gives a higher abstraction level to the *User Application* in form of handlers.

5.3.1. IO Control Module

The IO control related source files, called *control_IO.h* and *control_IO.c*, can be found in the */src/drivers/* folder. The module handles the port initializations for the physical HW platform e.g. LEDs, push-buttons, buzzer. It can set the state of the LEDs and read the status of the selected push-buttons.

```
void vCio_InitIO(void);
void vCio_SetLed(U8 biLedNum);
void vCio_ClearLed(U8 biLedNum);
BIT gCio_GetPB(U8 biPbNum);
```

5.3.2. Programmable Counter Array Driver

The PCA related source files, called *pca.h* and *pca.c*, can be found in the */src/driver/* folder. The module initializes the PCA so as to create beeping sounds on the buzzer. The time-base source of the PCA counter can be selected. Interrupts can be generated when the lower byte of the counter overflows. PWM-mod cycle length can be also selected to modify the frequency of the tweeting sound.

```
void vPca_InitPcaTmr(U8 biPulseSelect, U8 biPcaTmrItEnable, U8 biCycleLengthSelect);
```

5.3.3. SPI Driver

The SPI related source files, called *spi.h and spi.c*, can be found in the /src/driver/ folder. To enable the SPI interface the SPI port has to be enabled and associated to the crossbar. The directions of the SCK, MISO and MOSI ports have to be configured properly on the IO port. Finally, the default states of the pins have to be set correctly. Since several devices can be connected to the same SPI bus, the NSEL pin of the selected device is activated during communication. Since the commands to be sent to the API are sequences of bytes, the module has to be able to send and receive continuous byte stream. There are some cases when either reading a single byte directly from the MISO or writing specified number of bits directly to the MOSI is necessary. In order to cover these kinds of cases, “bitbang” read/write methods have been also implemented.

```
U8 bSpi_ReadWriteSpi1(U8 biDataIn);
void vSpi_EnableSpi1(void);
void vSpi_DisableSpi1(void);
void vSpi_ClearNselSpi1(U8 biSelectDevice);
void vSpi_SetNselSpi1(U8 biSelectDevice);
void vSpi_WriteDataSpi1(U8 biDataInLength, U8 *pabiDataIn);
void vSpi_ReadDataSpi1(U8 biDataOutLength, U8 *paboDataOut);
U8 bSpi_ReadByteBitbangSpi1(void);
void vSpi_WriteBitsBitbangSpi1(U8 biDataIn, U8 biNumOfBits);
```

5.3.4. Timer Peripheral Driver

The timer related source files, called *timer.h* and *timer.c*, can be found in *the /src/drivers/* folder. The module handles two 16-bit timers, timer2 and timer3. The most accurate timing interval can be calculated from the frequency of the system clock which is generally 24.5 MHz. External clock sources can be selected as timer input and moreover the required timing frequency can be adjusted thoroughly with several different prescalers. In general, the timer settings belonging to the so-called heart-beat frequency of 1 kHz (1 ms) are prepared. Using the timer with the 1 ms settings, timeouts multiple of 1 ms can be easily implemented. Timer related operations can give possibilities to start or stop counting. Additionally, interrupts can be generated when the low byte of the timer overflows. Timers can also be checked whether get expired or not.

```
void   vTmr_InitTmr2(void);
void   vTmr_StartTmr2(U8 biPrescaler, U16 wiPeriod, U8 biItEnable, U8 biExtClkSel);
BIT    gTmr_Tmr2Expired(void);
void   vTmr_StartTmr3(U8 biPrescaler, U16 wiPeriod, U8 biItEnable, U8 biExtClkSel);
UU16   wwTmr_GetTmr3(void);
BIT    gTmr_Tmr3Expired(void);
```

5.3.5. Human-Machine Interface Module

The HMI related source files, called hmi.h and hmi.c, can be found in the */src/driver/* folder. In order to use this module, the required handlers need to be initialized at the very beginning of the program. Checking the status of the various hardware components require to have a common cyclic mechanism. A 1 ms interrupt based cycle is vital to be running in the background to serve the different handlers.

```
HANDLERS

//Start the push button handler
void vHmi_InitPbHandler(void);

// Start the Led handler
void vHmi_InitLedHandler(void);

// Start the Buzzer Handler
void vHmi_InitBuzzer(void);
```

By using the LED handler, states of LEDs can be set and cleared either separately or together.

```
void vHmi_InitLedHandler(void);
void vHmi_ChangeLedState(eHmi_Leds qiLed, eHmi_LedStates qiLedState);
void vHmi_ChangeAllLedState(eHmi_LedStates qiLedState);
void vHmi_LedHandler(void);
void vHmi_ClearAllLeds(void);
```

To manipulate the LEDs' behavior dynamically, *eHmi_LedStates* needs to be used.

```
typedef enum _eHmi_LedStates
{
  eHmi_LedOff_c          = 0x00, /**< Led is in off state */
  eHmi_LedStdBy_c        = 0x01, /**< Led waits for state change */
  eHmi_LedOn_c           = 0x10, /**< Led is in on state */
  eHmi_LedBlink2Hz_c     = 0x20, /**< Led blinks with 2Hz */
  eHmi_LedBlink1Hz_c     = 0x30, /**< Led blinks with 1Hz */
  eHmi_LedBlinkHalfHz_c  = 0x40, /**< Led blinks with 0.5Hz */
  eHmi_LedBlinkOnce_c    = 0x50, /**< Blinks led once */
  eHmi_LedBlinkWait_c    = 0x55, /**< Blinks led once */
} eHmi_LedStates;
```

By using the button handler, statuses of the push-buttons can be read. Even if more button events happened simultaneously, they can be stored to be handled later. The last pushed button event is always available first amongst the unhandled events.

```
void vHmi_InitPbHandler(void);
BIT gHmi_PbIsPushed(U8 *boPbPushTrack, U16 *woPbPushTime);
BIT gHmi_IsPbUnHandled(void);
U8  bHmi_PbGetLastButton(U16 *woPbPushTime);
void vHmi_PbHandler(void);
```

By using the buzzer related sub-interface, state of buzzer can be changed to the required one.

```
void vHmi_InitBuzzer(void);
void vHmi_ChangeBuzzState(eHmi_BuzzStates qiBuzzState);
void vHmi_BuzzHandler(void);
```

To manipulate the buzzer's behavior dynamically, eHmi_BuzzStates needs to be used.

```
typedef enum _eHmi_BuzzStates
{
  eHmi_BuzzOff_c       = 0x00, /**< Buzz is in off state */
  eHmi_BuzzStdBy_c     = 0x01, /**< Buzz waits for state change */
  eHmi_BuzzOn_c        = 0x10, /**< Buzz is in on state */
  eHmi_Buzz2Hz_c       = 0x20, /**< Buzz 2Hz */
  eHmi_Buzz1Hz_c       = 0x30, /**< Buzz 1Hz */
  eHmi_Buzz0Hz5_c      = 0x40, /**< Buzz 0.5Hz */
  eHmi_Buzz0Hz25_c     = 0x41, /**< Buzz 0.25Hz */
  eHmi_BuzzOnce_c      = 0x50, /**< Buzz once */
  eHmi_BuzzOnceAndCont_c = 0x51, /**< Buzz once and continue buzzing*/
  eHmi_BuzzWait_c      = 0x55, /**< Waits for buzz once */
} eHmi_BuzzStates;
```

5.4. Empty Project Structure

An empty project structure can be found along with sample projects on the silabs.com website for reference.

The empty project has been created to help users starting to write their custom firmware with preconfigured project files, common directory structure, and Silicon Labs drivers.

The project follows the convention for directory structure introduced in the sample projects. It contains driver modules for the radio and MCU peripherals as well as a default MCU initialization procedure.

The *main* function in the empty project is as follows:

```
void main(void)
{
 // Initialize the Hardware and Radio
 vInitializeHW();

 while (TRUE)
 {
  // Write your code HERE.
 }
}
```

User defined init functions should be placed before the *while* loop. The *vInitializeHW()* looks as follows:

```
void vInitializeHW(void)
{
  // Initialize the MCU peripherals
  vPlf_McuInit();

  // Initialize the Radio
  vRadio_Init();
}
```

The *radio_config.h* configuration should be generated, edited then added to project in order to enable the *vRadio_Init()* function to initialize the radio chip. The *vPlf_McuInit()* shall be modified according to the hardware used.

The additional header files should be included in the common *bsp.h* located in src/drivers directory.

The *bsp.h* file already contains included headers:

```
/*---------------------------------------------------------------------*/
/*          Application specific includes                          */
/*---------------------------------------------------------------------*/

#include "compiler_defs.h"
#include "platform_defs.h"
#include "hardware_defs.h"

#include "..\application\application_defs.h"

#include "spi.h"

#include "..\application\radio_config.h"
#include "..\application\radio.h"

#include "..\drivers\radio\radio_hal.h"
#include "..\drivers\radio\radio_comm.h"

#ifdef SILABS_RADIO_SI446X
#include "..\drivers\radio\Si446x\si446x_api_lib.h"
#include "..\drivers\radio\Si446x\si446x_defs.h"
#include "..\drivers\radio\Si446x\si446x_nirq.h"
#include "..\drivers\radio\Si446x\si446x_patch.h"
#endif

#ifdef SILABS_RADIO_SI4455
#include "..\drivers\radio\Si4455\si4455_api_lib.h"
#include "..\drivers\radio\Si4455\si4455_defs.h"
#include "..\drivers\radio\Si4455\si4455_nirq.h"
#endif
```

Description of the header files:

compiler_defs.h—Contains compiler related definitions in order to provide a uniform way to use non-ANSI C keywords, such as BIT, XDATA, SEGMENT_VARIABLE and more.

platform_defs.h—Platform definitions, in this header the type of hardware, MCU can be defined.

hardware_defs.h—Definitions can be found in this header that closely related to the actual hardware, for example, which pins connected to LEDs, push-buttons, etc.

application_defs.h—Definitions for the given application.

spi.h—Serial Peripheral Interface driver header file. Contains functions to access and utilize the MCUs SPI interface.

radio_config.h—Radio configuration header. This file is generated by the Wireless Development Studio (WDS). The user may want to add custom configurations to this file.

radio_hal.h—Common hardware abstraction layer for both Si446x and Si4455 radio families.

radio_comm.h—Common communication layer for both Si446x and Si4455 radio families.

The next group of header includes depends on the actual radio chip defined in *platform_defs.h*.

5.4.1. Porting the Project for another MCU

The porting of an example project to an MCU of choice can be done easily thanks to the layered approach of the project structure. This reduces the effort required to compile the code for other architecture, as only the low-level functions must be modified.

The following drivers shall be modified:

- compiler_defs.h, hardware_defs.h, platform_defs.h, application_defs.h
  This header files contains definitions for the 8051 architecture and the SiLabs hardware platform. This may be modified according to the new architecture and hardware.
- spi.c, spi.h
  The SPI driver module shall be supplied to enable the communication with the radio.
- radio_hal.c, radio_hal.h
  The radio hardware abstraction layer may be adjusted, as the GPIOs, nIRQ and SDN pins are defined in this file.

The above mentioned files may not cover all requirements for porting the project to other MCU, as it depends on what is to be ported and which other drivers are used by the project.

The compiler toolchain setup, the appropriate startup codes, and linker scripts are out of the scope of this chapter; the user is responsible for providing them as appropriate for the given architecture.

NOTES:

Simplicity Studio

One-click access to MCU tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

www.silabs.com/simplicity

MCU Portfolio
www.silabs.com/mcu

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com