TRAINING MANUAL

Atmel AT7159: Designing a Wireless Sensor in ASF on the SAM4L-EK - Thermostat Example

ASF PROJECT DOCUMENTATION

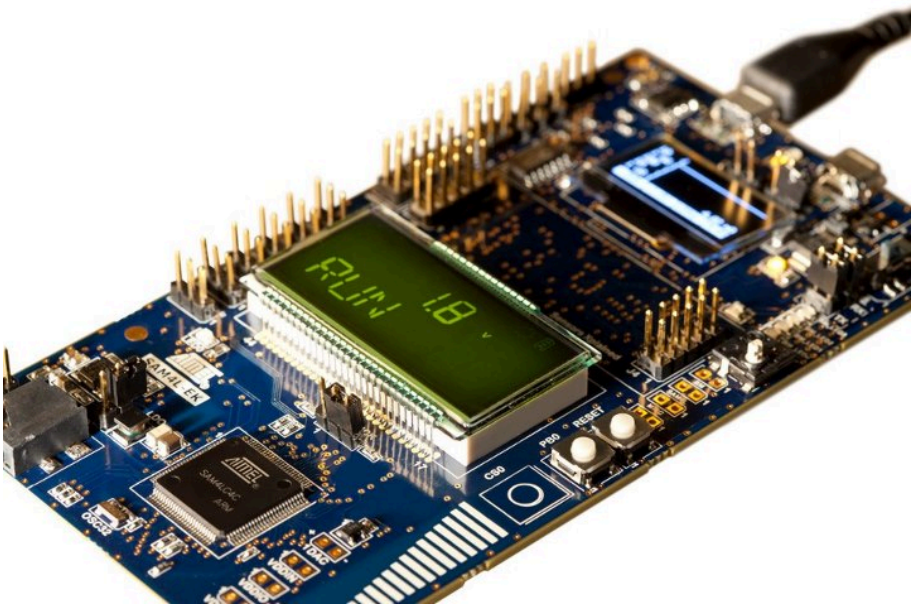


Table of Contents

1. Introduction

The Internet of Things (IoT) is a paradigm where people interact with everyday objects through the Internet. An IoT solution will typically feature a number of smart sensors interconnected in a low-power wireless network. In this training module we will develop an IoT application following efficient system design techniques recommended by Atmel experts. Note that the lessons learned will help you build applications in a wide range of domains, for example home automation, logistics, industrial control, smart metering, smart cities, smart agriculture, smart farming, etc.

Specifically, in this hands-on session we will work on a home automation solution where a thermostat and a heating, ventilation and air conditioning (HVAC) system communicate in a wireless network. These two elements are described below:

- A thermostat is a smart sensor that provides information to the HVAC system. An example is shown in Figure 1-1: An example of thermostat. In this training hands-on session you will design a thermostat prototype using Atmel Software Framework (see ASF User Guide) and SAM4L-EK (see Prerequisites).

INFO We will use a light sensor to simulate the temperature sensor. This is mainly because it is easy to quickly change the light value, e.g. by moving your finger above the sensor.

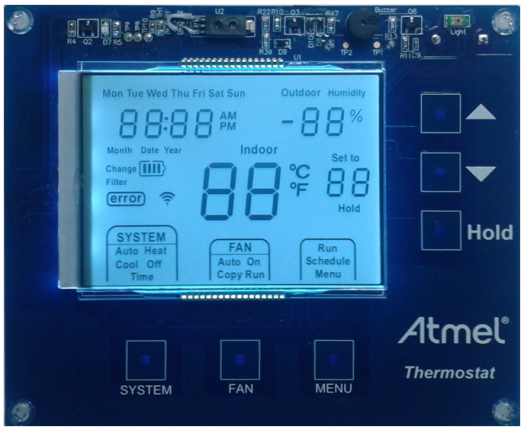
Figure 1-1. An example of thermostat

- The HVAC system has a head unit that receives the information provided by the thermostat. This information is used to maintain the temperature, humidity, etc. near a desired point.

INFO In this training session the head unit is already built and waiting to receive information. When your thermostat is ready, your data will be printed on the classroom screen!

Our training application is a simplification of a thermostat reference design developed by Atmel, which is shown in Figure 1-2: Atmel thermostat reference design

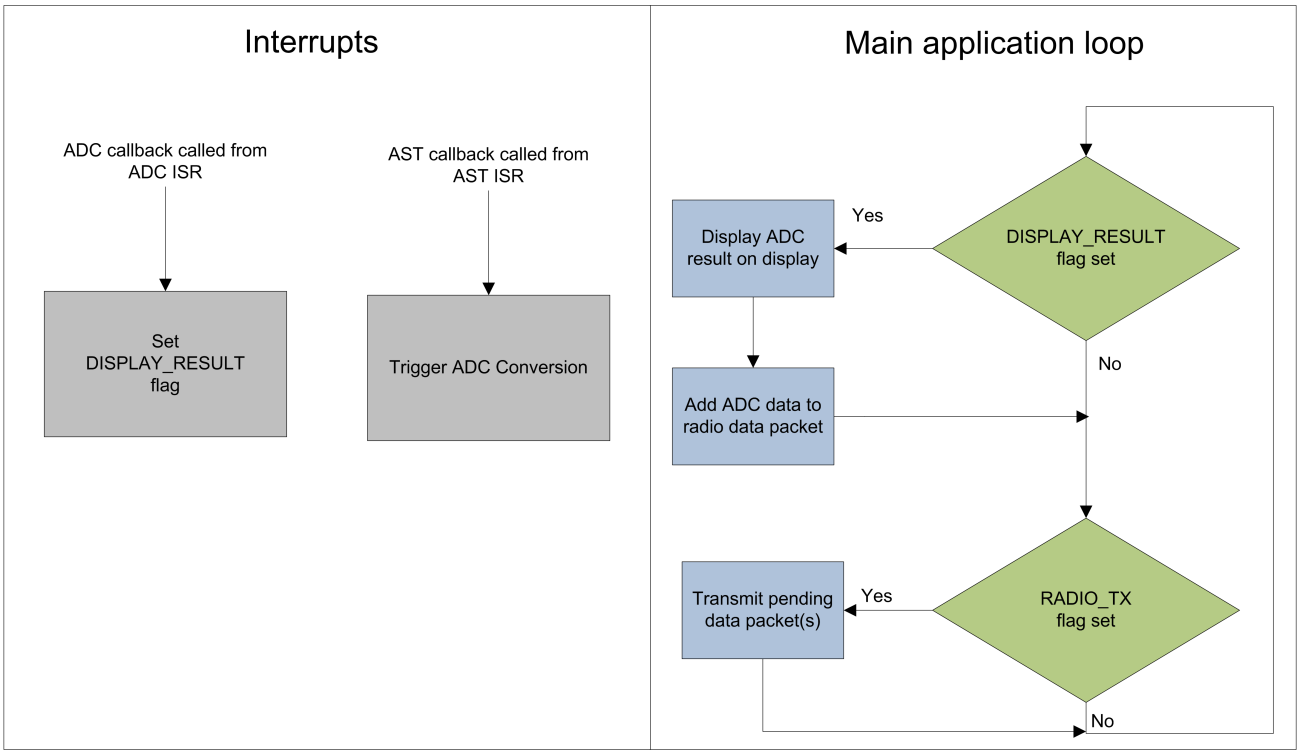
Figure 1-2. Atmel thermostat reference design



1.1 Target Application Description

In this section we give you a more detailed preview of the application we are going to build in this training session. Figure 1-3: Application flow chart shows a flow diagram of the completed application, where you can see how the main application and the interrupts interact. The ADC sample buffer is filled each second in the AST interrupt routine. When the data is ready, the main application is signaled so that the data is displayed on the LCD and transmitted to the head-unit through the wireless network.

Figure 1-3. Application flow chart



The primary focus of this session is on the wireless features and, therefore, the first chapter (Wireless Communication) starts from a demo application that reads from the light sensor and shows the value on the LCD display. In that chapter we show how to add wireless communication between the demo application and a pre-made head-unit by using the wireless MAC stack included in Atmel Software Framework. The following chapters (Develop a Basic Application (OPTIONAL) and Light Sensor Acquisition (OPTIONAL)) show how to build the basic application from an empty project. You can cover them during the hands-on session if you have time, or later if you want to learn more!

1.2 Atmel Design Best Practices: Guaranteed Embedded Design Success!

This application has been developed using the integrated development platform Atmel Studio 6.1. For quick hardware and software prototyping we have used Atmel Software Framework 3.8.1 and the SAM4L-EK.

The embedded software developers working on this project used Git, a distributed revision control and source code management system. In addition, we used a private GitHub clone as a code management and collaboration solution for the project development.

Our software development is based on the Scrum methodology, an iterative and incremental agile software development framework for managing software projects. We use Jira and GreenHopper for issue tracking and agile project management.

The project was documented using Doxygen, a free tool to generate documentation from the source code. This document was built using a custom XSL transform to convert Doxygen XML into the DocBook open document standard, and then into a PDF via a second XSL transform and the open source Apache FOP tool.

1.3 Prerequisites

This training session requires the following software and hardware:

- Atmel Studio 6.1 (build 2562)
- Atmel Software Framework 3.8.1
- SAM4L-EK
- RZ600-RF231
- USB Micro B cable

1.4 Icon Key Identifiers

Icons are used to identify different assignment sections and reduce complexity. These icons are:

INFO Delivers contextual information about a specific topic

TIPS Highlights useful tips and techniques

TODO Highlights objectives to be completed

RESULT Highlights the expected result of an assignment step

WARNING Indicates important information

EXECUTE Highlights actions to be executed

2. Wireless Communication

In this chapter we will first cover the software and hardware setup: you will start from a basic demo application that reads from a light sensor and shows the result on the LCD display, which more or less corresponds to a smart sensor (thermostat) without wireless connectivity. After that, we will show you how to connect the smart sensor (thermostat) and the head-unit (HVAC), and share information through a wireless network. Specifically, we will use the AT86RF231[1] 2.4GHz radio transceiver boards from the RZ600 kit[2] for transmitting and receiving data over a wireless link. We will focus on setting up the MAC Stack included in Atmel Software Framework for the SAM4L-EK, but the design process is general and applies to any supported device.

The overview of this chapter is as follows:

- Software and Hardware Setup
- Introduction to the Wireless MAC Stack
- Add and Setup Wireless MAC Service
- Sensor Data Transmission
- Summary

2.1 Software and Hardware Setup

In this section we show how to set up the software and hardware. In addition, we will guide until you have built and run the basic application.

2.1.1 Software Setup

Atmel Studio 6[3] is the integrated development platform (IDP) for developing and debugging Atmel ARM Cortex-M processor-based and Atmel AVR microcontroller applications. Atmel Studio 6 IDP gives you a seamless and easy-to-use environment to write, build and debug your applications written in C/C++ or assembly code. Atmel Studio 6 supports all 8- and 32-bit AVR, the new SoC wireless family, SAM3 and SAM4 microcontrollers, and connects seamlessly to Atmel debuggers and development kits.

Atmel Studio includes some features designed to further enhance your productivity:

- Atmel Gallery[4] is an online app store built into Studio 6, allowing you to download both in-house and third-party development tools and embedded software.

Figure 2-1. Atmel Gallery

- Atmel Training has recently started contributing to Atmel Gallery by adding free extensions with hands-on and other exciting material. We want to make sure that your embedded design is successful and, therefore, we have chosen to make our training catalog more accessible to the public.

Figure 2-2. Atmel Training

[1] http://www.atmel.com/devices/at86rf231.aspx
[2] http://www.atmel.com/tools/RZ600.aspx
[3] http://www.atmel.com/microsite/atmel_studio6/
[4] http://gallery.atmel.com/

- Atmel Spaces[5] is a collaborative workspace where you can securely share embedded design and track progress of projects with your peers.

Figure 2-3. Atmel Spaces

INFO The version that we are going to use is Atmel Studio 6.1 (build 2562)

TODO Install the training extension

In the training session you will be supplied a training extension `Atmel-AT7159-x.y.z.vsix`. You can install it by double-clicking on it. This training extension will install three projects:

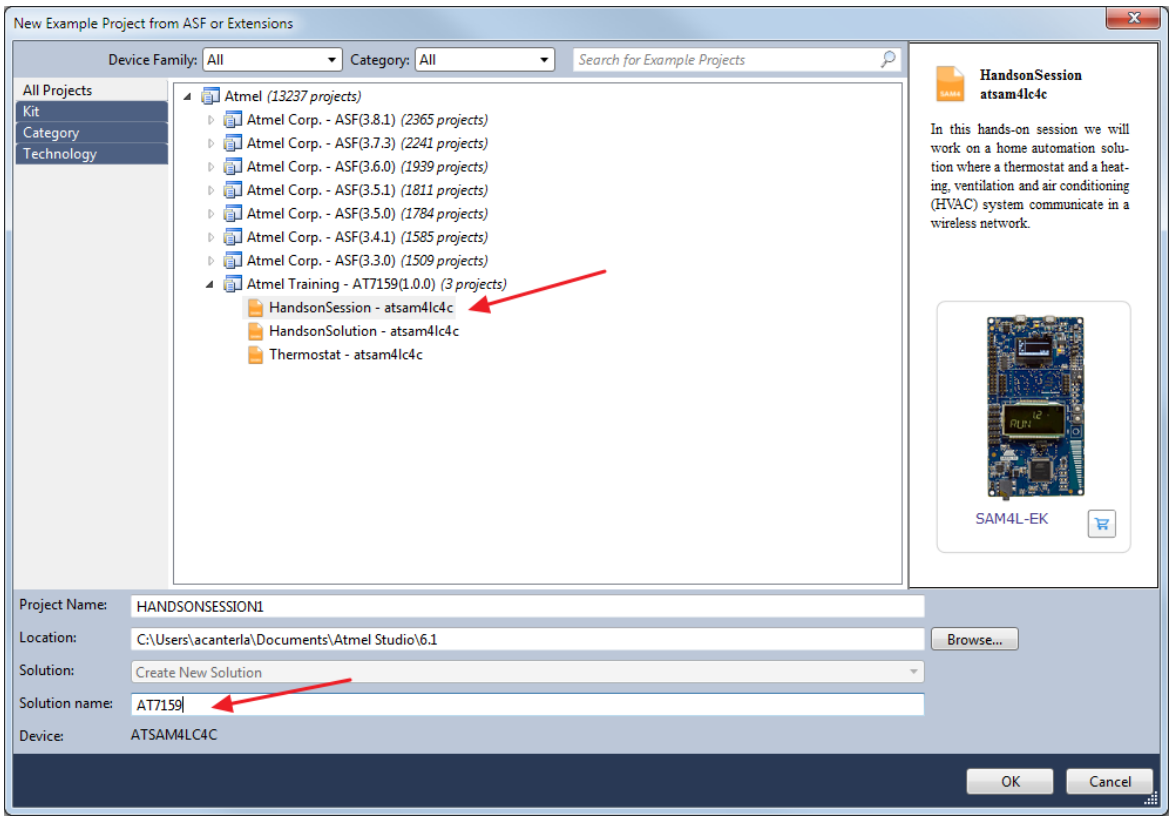
- *HandsonSession*: this project delivers the hands-on document, the basic application and other files that are required to complete this training session.
- *HandsonSolution*: this project provides the completed assignment for this training session
- *Thermostat*: this project provides the basic application that we use as starting point

TODO Open the installed project *HandsonSession*:

After the training extension has been installed, you can open Atmel Studio, select `File > New > Example Project..` and select the project *HandsonSession* as shown in Figure 2-4: Loading the training project HandsonSession installed by the extension. Select *AT7159* as the name of the solution that will contain the project. After that you can click on *OK*.

[5] http://spaces.atmel.com/gf/

Figure 2-4. Loading the training project HandsonSession installed by the extension



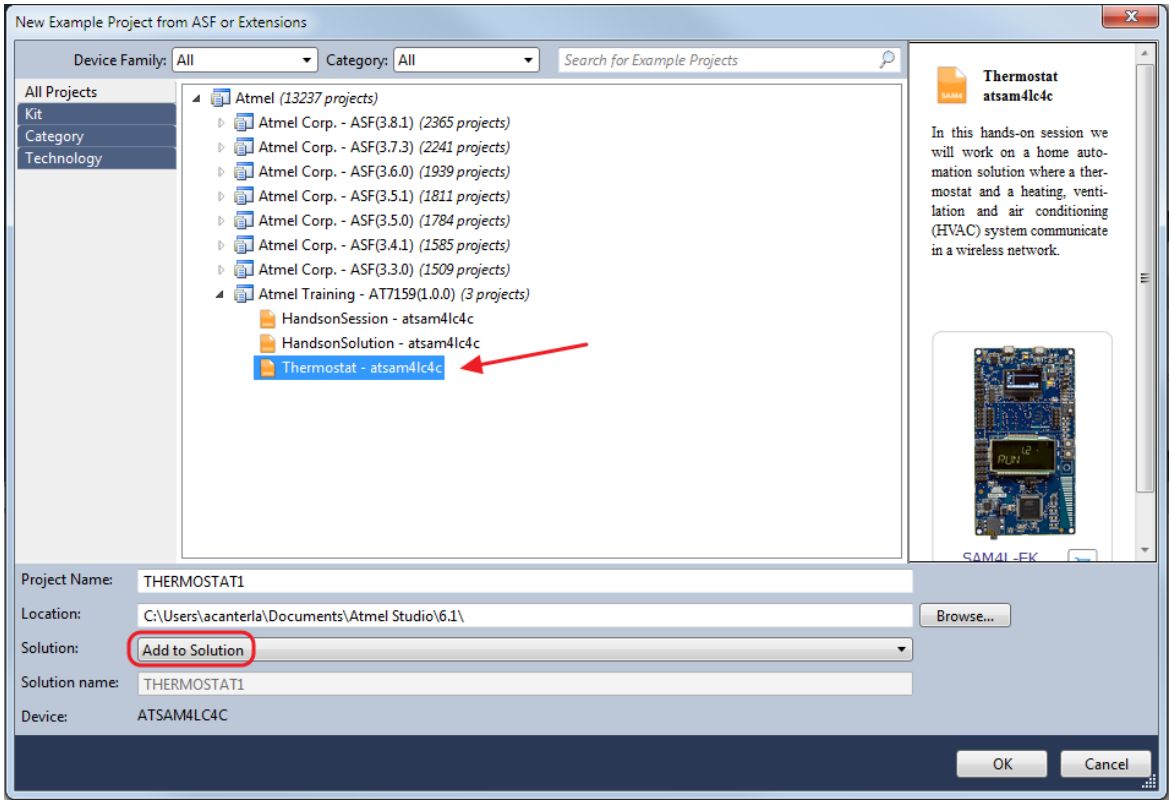
RESULT Atmel Studio will create a solution called *AT7159* and open the project *HandsonSession*.

WARNING We will *add* a new project to the existing solution. You need to make sure you select "Add to solution" in the next TODO point.

TODO Add the project *Thermostat* to your solution

Select `File > New > Example Project..` and select the project *Thermostat* as shown in Figure 2-5: Loading the training project Thermostat installed by the extension.

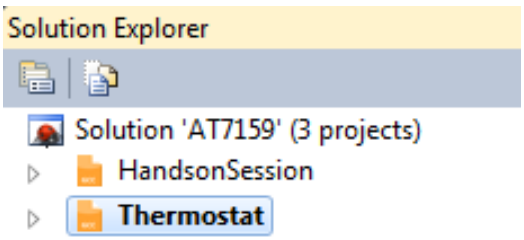
Figure 2-5. Loading the training project Thermostat installed by the extension



RESULT

Atmel Studio will add the project *Thermostat* to the solution *AT7159*, as shown in This is the project that contains the basic application that we use as starting point in this training session. This means that this will be the project where you will be developing the smart-sensor (thermostat) application.

Figure 2-6. Thermostat project has been added to the solution



TIPS

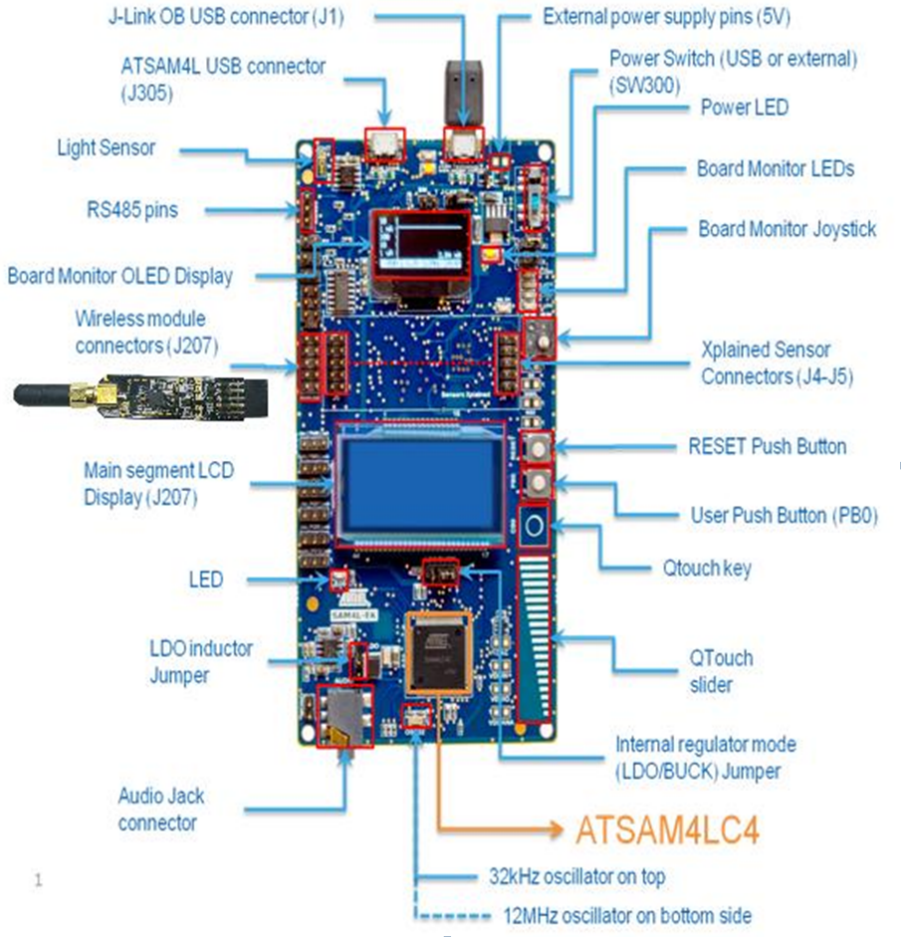
When there are two projects in a solution, we need to select one of them as the startup-project, i.e. the one that will be compiled and programmed to the device. Check that the "Thermostat" project is shown with bold text in the Solution Explorer. Otherwise right-click it and select *Set as StartUp project*.

2.1.2 Atmel SAM4L-EK

The main board used in this hands-on session is the SAM4L-EK, which is shown in Figure 2-7: SAM4L-EK. This board features an embedded debugger, dedicated circuitry to measure the power consumption of your application,

LCD, USB, capacitive touch functionality and much more. The board also offers an expansion header to connect Atmel RZ600-RF transceivers so that you can easily develop your wireless product prototype. In the product page[6] (selecting the "Documents" tab) we get access to two important documents that can help you with your design: the user guide[7] and the schematics[8]

Figure 2-7. SAM4L-EK



A key feature of the SAM4L-EK is the **embedded debugger**. This is a debugger that is populated on the board, making it possible to debug code on the target device without any external hardware. The only connection needed to get started with the SAM4L-EK is a micro USB cable to a computer running Atmel Studio 6.1. The embedded debugger uses a standard interface, CMSIS-DAP (an open debugging interface made by ARM). This allows not only Atmel, but also third parties to provide debug support for the kit.

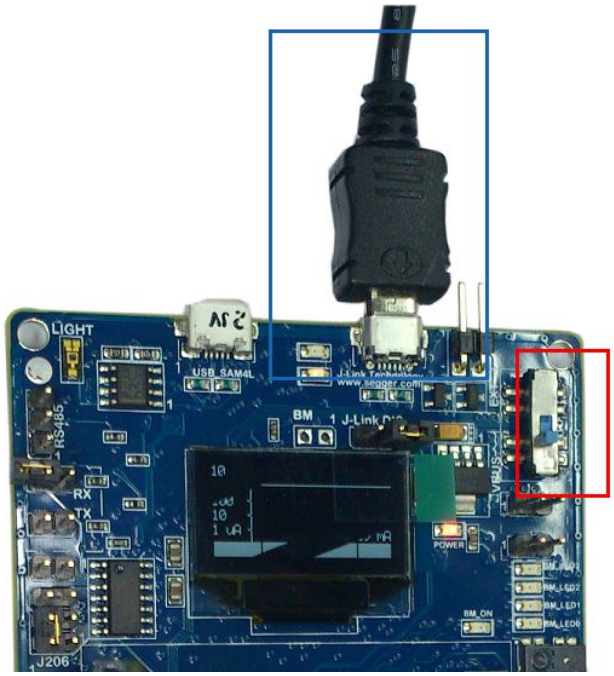
To further allow for rapid development, SAM4L-EK includes both a USB CDC (Virtual COM port) connected to a USART on the target MCU. This allows for getting data easily out of the target MCU for additional processing or verification on the debugging platform. Examples of this usage can be ADC sample plotting, logging debug data or simulating external systems during debugging.

TODO Connect the board

The first thing we need to do is to connect the USB cable to your computer and the other end to the board as shown in Figure 2-8: SAM4L-EK connection and power up (blue square).

[6] http://www.atmel.com/tools/SAM4L-EK.aspx?tab=overview
[7] http://www.atmel.com/Images/Atmel-42026-ATSAM4L-EK-User-Guide_Application-Note_AVR32850.pdf
[8] http://www.atmel.in/Images/doc42027_SAM4L-EK_Design_Documentation.PDF

Figure 2-8. SAM4L-EK connection and power up



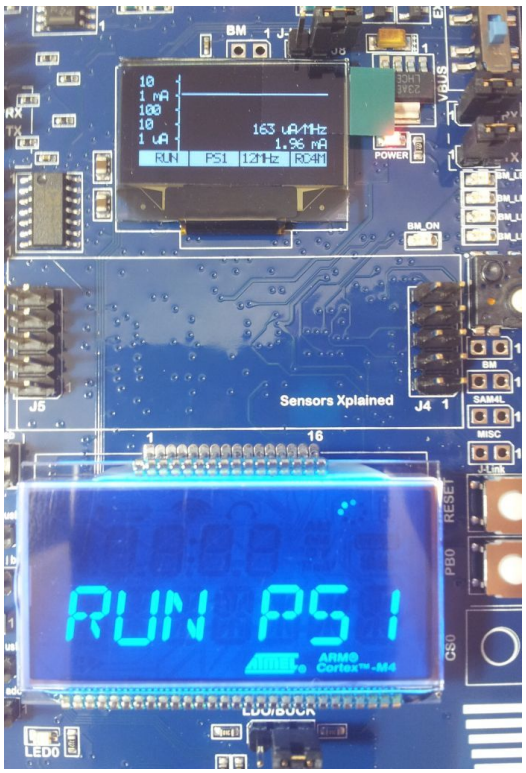
TIPS The USB cable is plugged into the J-Link connector.

TODO Power up the board

After that we need to put the power switch into the position "VBUS", as shown in Figure 2-8: SAM4L-EK connection and power up (red square).

RESULT The board will be powered up and the default demo application will be running. This is shown in Figure 2-9: Default demo application in SAM4L-EK. The drivers for the embedded debugger on the SAM4L-EK will be installed automatically when connecting the board.

Figure 2-9. Default demo application in SAM4L-EK



TIPS The small screen shows the energy consumption of the board. This monitoring application is running in the board controller and will work independently of the software in the SAM4L. However, note that the display of the operating mode requires handshake between the board controller and the SAM4L.

2.1.3 Light Sensor Setup

We need to establish a physical connection between the sensor output and the ADC pin.

TODO Connect the light sensor output to ADC pin

The user guide[9] for the SAM4L-EK provides us with the following information:

- The SAM4L-EK is equipped with one light sensor connected to one of the ADC channels (on pin PB05). The light sensor I/O mapping presented in the user guide is reproduced in Table 2-1: Light sensor I/O mapping.

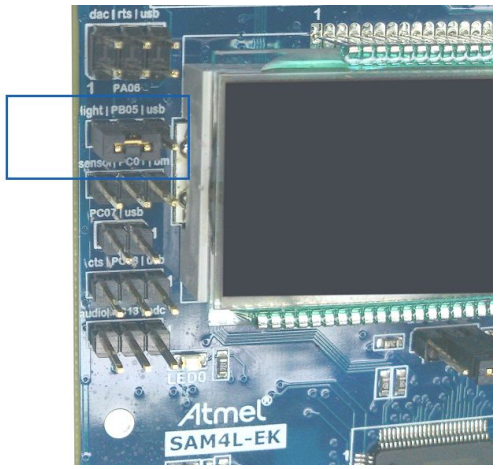
Table 2-1. Light sensor I/O mapping

GPIO	Feature	Jumper
PB05	ADC Channel	Close J101.2 to J101.3

Therefore, in order to enable the light sensor we need to **close the jumper J101.2 to J101.3**. This is shown in Figure 2-10: Jumper Connected.

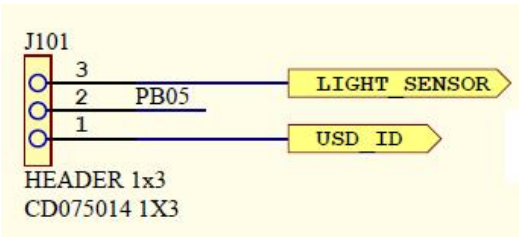
[9] http://www.atmel.com/Images/Atmel-42026-ATSAM4L-EK-User-Guide_Application-Note_AVR32850.pdf

Figure 2-10. Jumper Connected



Note that this information is also shown in the schematics[10] for the SAM4L-EK. An extract is shown in Figure 2-11: Light sensor jumper schematic

Figure 2-11. Light sensor jumper schematic



2.1.4 Building and Running the Basic Application

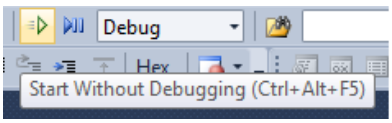
We are going to build and run the basic demo that we use as starting point in this training session.

TODO Build and run the application

We have two options: We can start a debug session on the board, where we will be able to break and follow the application flow, or we can simply program the compiled code to the controller and execute the application. This is shown in Figure 2-12: Start Without Debugging and Figure 2-13: Start Debugging and Break. We just want to verify our kit, so we will select the "Start Without Debugging".

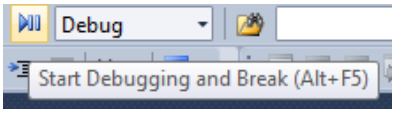
TIPS If you cannot find the "Start Without Debugging" icon, you can also select `Debug > Start Without Debugging`.

Figure 2-12. Start Without Debugging



[10] http://www.atmel.in/Images/doc42027_SAM4L-EK_Design_Documentation.PDF

Figure 2-13. Start Debugging and Break

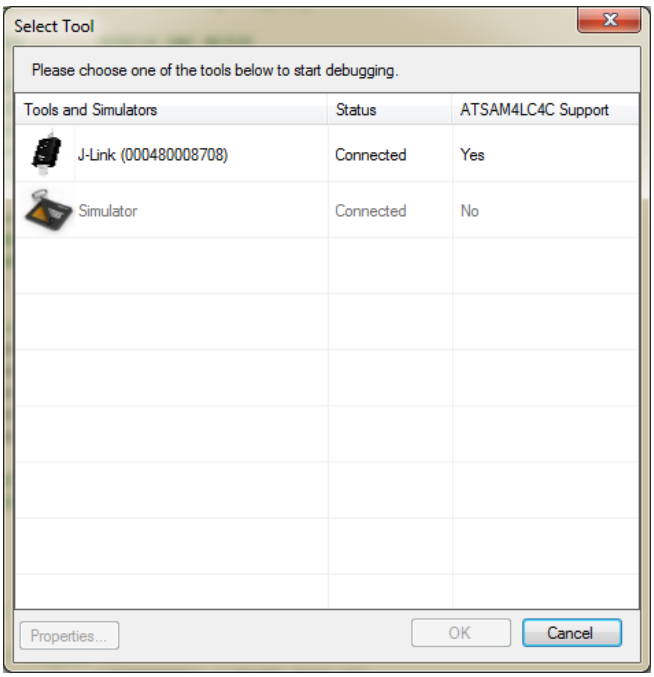


INFO

There could be a number of warnings when compiling the application. This is just because the functions that set, clear and check the flag variable `app_state_flags` are defined but not used yet.

We will now be presented with a tool select dialog that is used to select the correct tool to execute our software on. We will select our SAM4L-EK as shown in Figure 2-14: Select Tool.

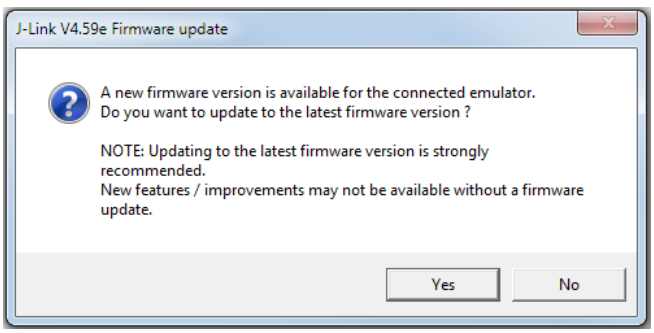
Figure 2-14. Select Tool



INFO

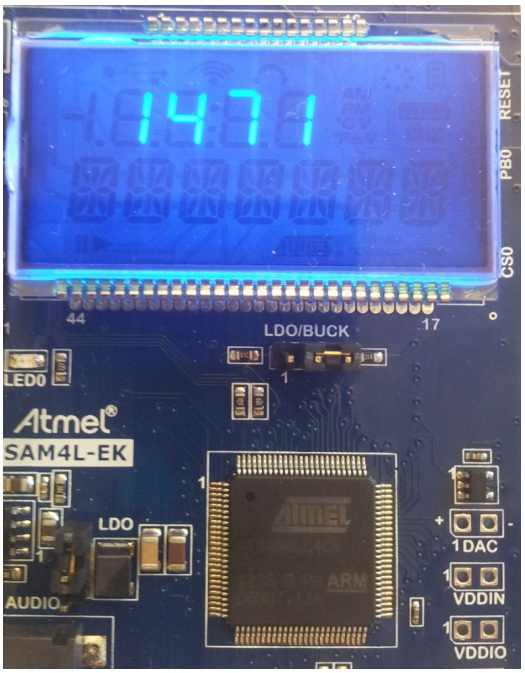
In some cases the firmware of the embedded debugger is out of date, and need to be updated. Atmel Studio will then show a prompt asking to update the firmware. This is shown in Figure 2-15: Firmware update dialog. If this is the case, simply press the "Upgrade" button to upgrade to the latest embedded debugger version.

Figure 2-15. Firmware update dialog



RESULT At this point you should see the value read from the light sensor updated on the display every second,as shown in Figure 2-16: Sample data on the LCD display. If you cover the light sensor (located on the top left corner of the board) with your finger, the value shown in the LCD display should change.

Figure 2-16. Sample data on the LCD display

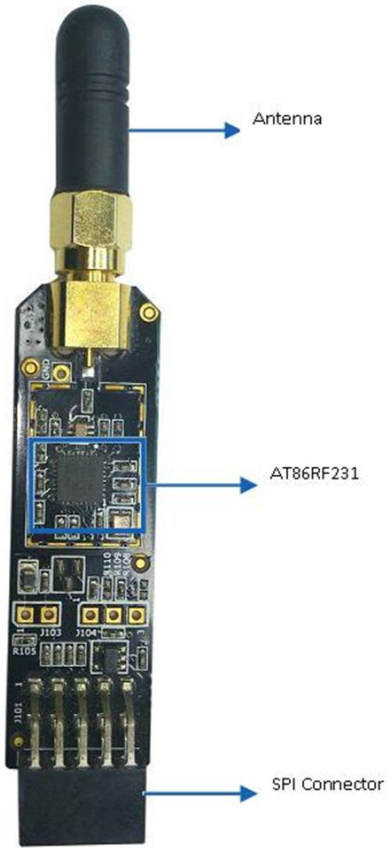


2.1.5 Atmel RZ600-RF231

RZ600 [11] is an FCC-certified kit which contains evaluation boards for Atmel AT86RF212 and the AT86RF23x families of RF transceivers. These cover the regional 700/800/900MHz and worldwide 2.4GHz ISM bands and can be used together with other Atmel evaluation kits. The kit also includes two USB adapters equipped with 32-bit AVR UC3A microcontrollers. In this training session we will use the RZ600-RF231[12] transceiver board, which is described in Figure 2-17: RZ600-RF231.

[11] http://www.atmel.com/tools/RZ600.aspx
[12] http://www.atmel.com/devices/at86rf231.aspx

Figure 2-17. RZ600-RF231



WARNING Make sure that you are using a RZ600-RF231 board by checking the part number of the transceiver chip mounted in the kit. You should be able to read "ATRF231".

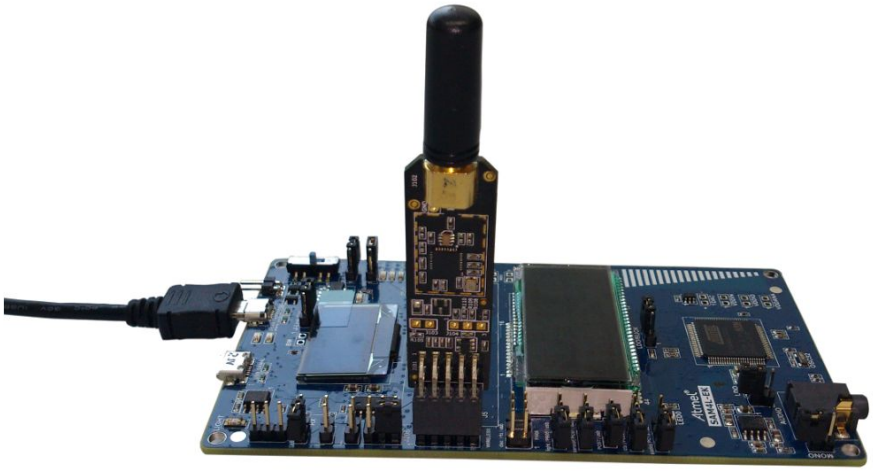
We are going to connect the RF transceiver to the SAM4L-EK. After that we will configure the board to be able to interface the transceiver correctly.

TODO Connect the RZ600-RF231 to the SAM4L-EK

The first step is to connect the RF transceiver to the SAM4L-EK. The SAM4L-EK user guide tells us that the transceiver connector should be plugged to the 10-pin header J207 in the SAM4L-EK. This is shown in Figure 2-18: RZ600-RF231 connected to SAM4L-EK.

TIPS The header is labelled as "Wireless" in the SAM4L-EK. Further, make sure you match the pin 1 in the connector with the pin 1 in the header.

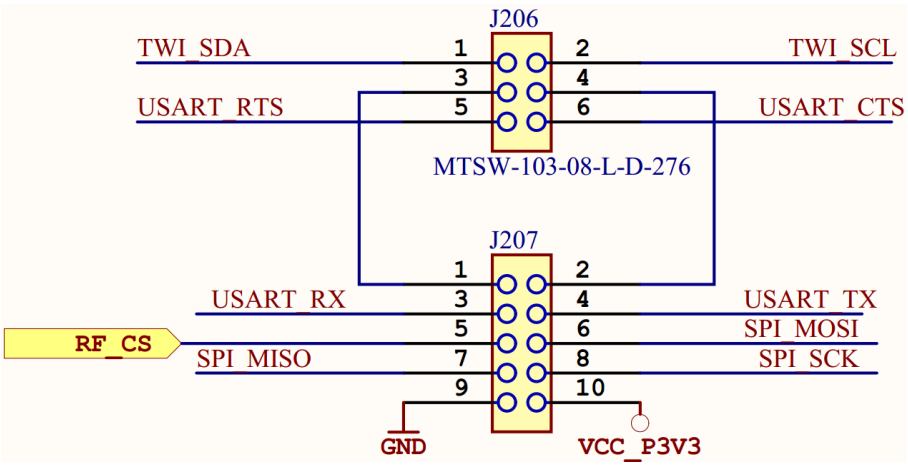
Figure 2-18. RZ600-RF231 connected to SAM4L-EK



TODO Choose reset line pin

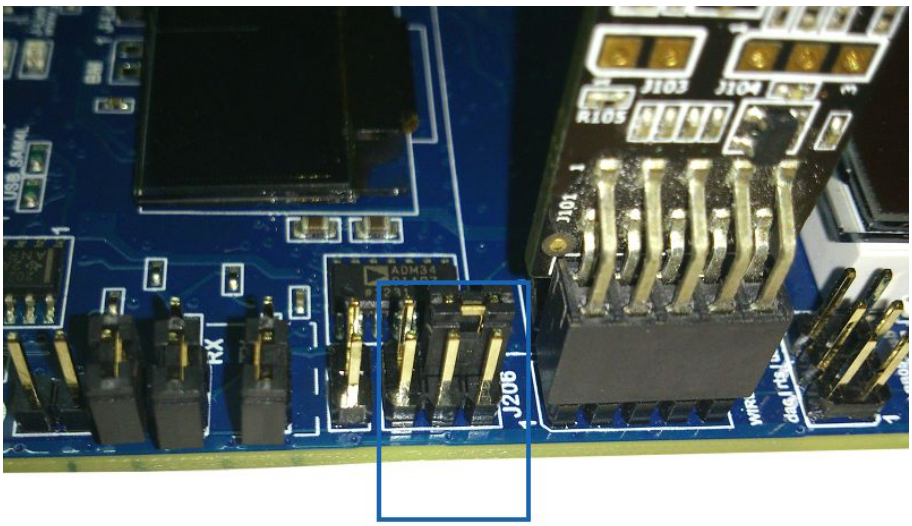
If we look at the SAM4L-EK schematics (see extract in Figure 2-19: Expansion header for wireless interface reproduces the concerned blocks), we see that we need to choose the reset line.

Figure 2-19. Expansion header for wireless interface



We want to choose the signal TWI_SDA (PB00) as reset line for the transceiver. Therefore, we need to **close the jumper J206.1 to J206.3**, as shown in Figure 2-20: J206 jumper connection.

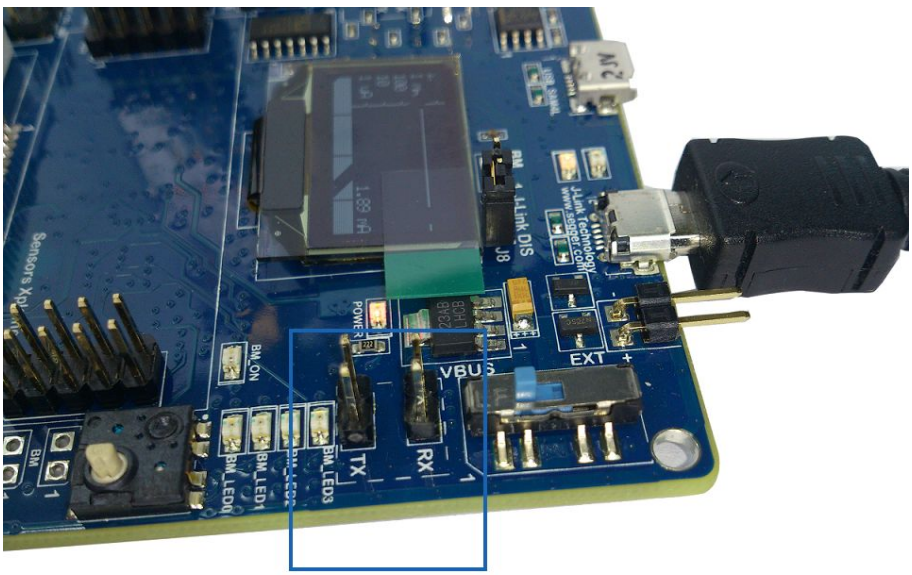
Figure 2-20. J206 jumper connection



TODO Disconnect USART interface provided by the embedded debugger

The pins 3 and 4 in the wireless header of the SAM4L-EK are shared with the USART interface offered by the embedded debugger. Therefore, we will need to **remove the jumper RX and TX** as shown in Figure 2-21: TX and RX jumpers should be removed. This is because the IO pin used as IRQ and SLEEPTR line for the transceiver are also used as Board Monitor lines for communicating with the embedded debugger.

Figure 2-21. TX and RX jumpers should be removed



WARNING The board has other pins called "TX" and "RX". Make sure you remove the pins that are next to the power switch, as shown in Figure 2-21: TX and RX jumpers should be removed.

2.2 Introduction to the Wireless MAC Stack

In this first section we will give a quick overview of the wireless MAC (*Media Access Control*) stack software package. A more detailed description of the MAC stack operation can be found in the document MAC Stack User Guide[13]

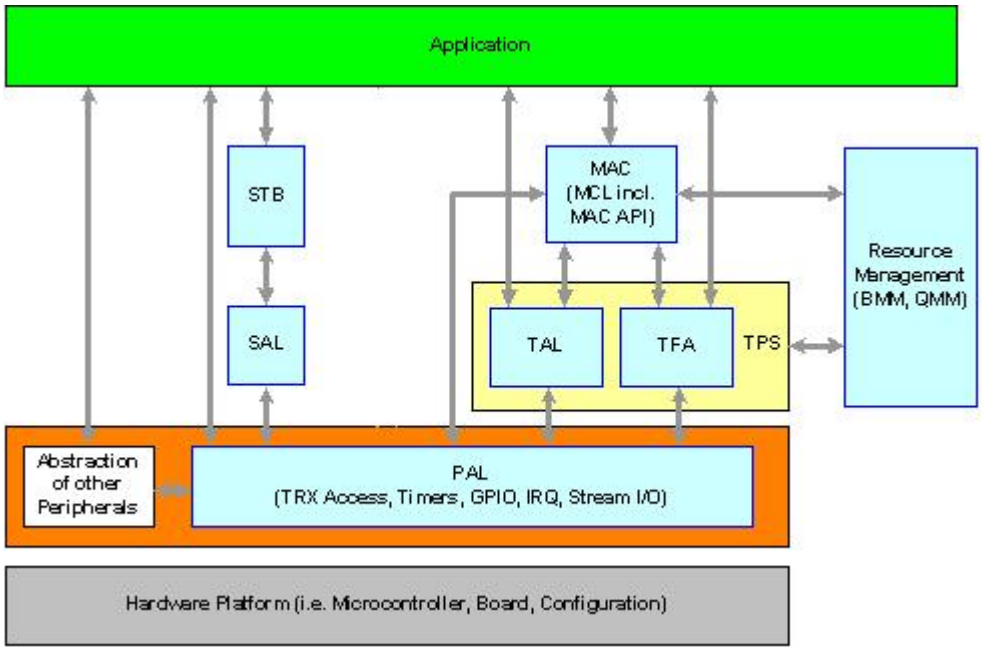
INFO The appendix Introduction to IEEE802.15.4 Standard gives a short introduction to the IEEE 802.15.4 standard, network topologies, etc.

The main MAC stack software consists of three layers starting from the bottom up:

- Platform Abstraction Layer - PAL
- Transceiver Abstraction Layer - TAL
- MAC Core layer - MCL

Figure 2-22: MAC Structure shows the implementation of the MAC layers in the MAC package.

Figure 2-22. MAC Structure



The **PAL** contains all platform-specific (i.e. MCU and board) functionality that is required by the MAC software package. It provides interfaces to the upper software stack layers and for each device platform a separate implementation exists within the PAL layer.

The **TAL** contains the transceiver specific functionality used for the IEEE 802.15.4-2006 MAC support. It provides interfaces to the MAC Core Layer which is independent from the underlying transceiver. In addition, the TAL API can be used to interface from a basic application. There exists exactly one implementation for each transceiver using transceiver-embedded hardware acceleration features. The TAL (on top of PAL) can be used for basic applications without adding the MCL.

The **MCL** abstracts and implements IEEE 802.15.4-2006 compliant behavior for non-beacon enabled and beacon-enabled network support. The MAC package will have several directories (e.g. `apps`, `pal`, `tal`, `mac`, etc.) each containing the source files implementing the functionality of the corresponding layer.

[13] http://www.atmel.com/Images/doc5182.pdf

2.3 Add and Setup Wireless MAC Service

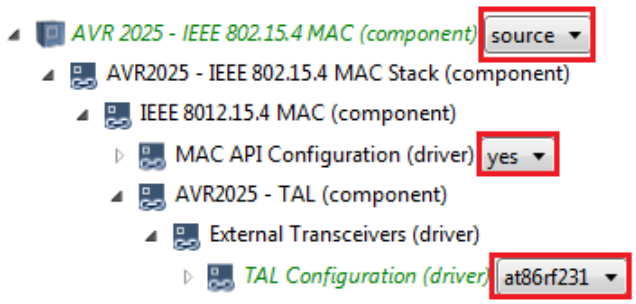
TODO Add the wireless MAC service using *ASF Wizard*

The basic application has already a number of Atmel Software Framework (ASF) drivers included: ADC, AST, LCD, etc. We will add the wireless MAC service (AVR 2025 - IEEE 802.15.4 MAC Configuration) using *ASF Wizard*. Please note that we need to select some settings **before** the MAC stack service is added to the project. If you need a description on how to add ASF drivers, please check Displaying Text on an LCD Display. The following settings are suitable for our application:

- AVR2025 - IEEE 802.15.4 MAC Configuration: Set to *source*.
- MAC API Configuration: Set to *yes*.
- TAL Configuration: Set to *at86rf231*, which is our RF transceiver.

The configuration is shown in Figure 2-23: Wireless MAC service selected.

Figure 2-23. Wireless MAC service selected



TODO Include MAC service in our application

All necessary header files are automatically included in *asf.h* after adding an ASF driver with *ASF Wizard*. However, the MAC stack is a third party ASF driver and we need to manually include the following header files in `main.c`:

```
#include "avr2025_mac.h"
#include "pal.h"
```

INFO In cases like this when we need to include a header file in our application, we can for example include it from *main.c* but never from *asf.h*. If you modify the latter, next time you include an ASF driver the file will be automatically generated and your changes will be discarded.

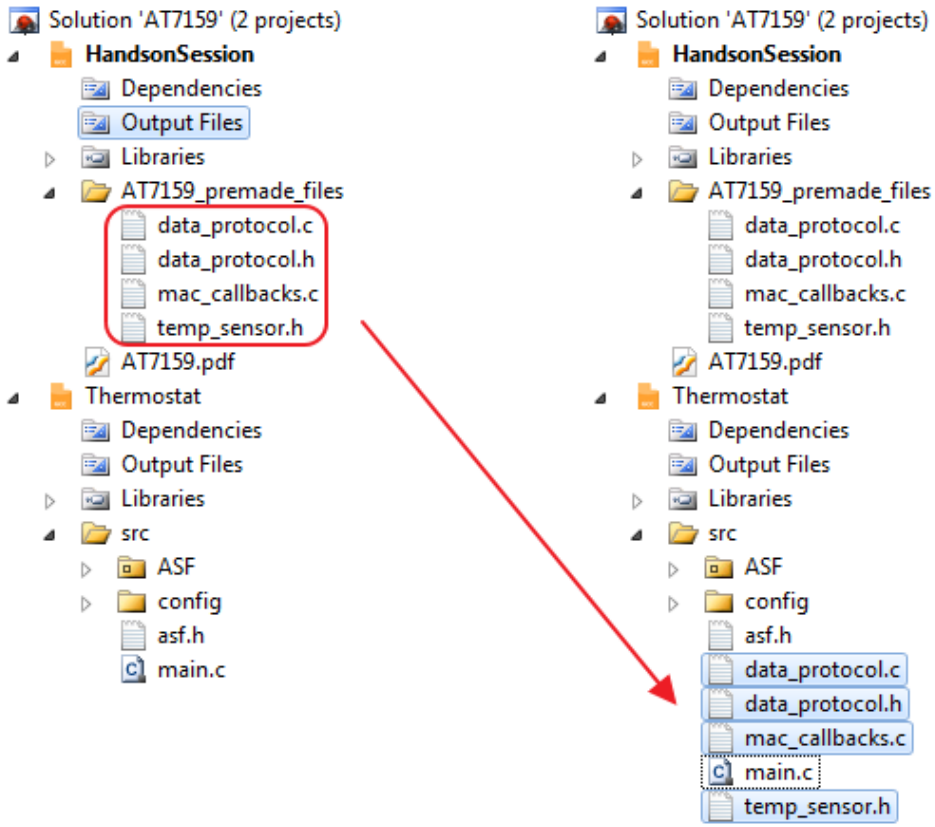
TIPS After your added the MAC service using *ASF Wizard*, a number of callback implementations were placed in the folder `3rd_party > wireless > avr8025_mac > source > mac > src`. For our

application we need to modify two of the default implementations. However, for simplicity, we will provide you the code for these two callback functions. The instructions are presented below.

TODO Add to the project the pre-made MAC callback functions and other helper files

From *Solution Explorer* open the folder *AT7159_premade_files* from the project *HandsonSession*. Copy these files to your *Thermostat* project by dragging and dropping the files into the *Solution Explorer* window. This is shown in Figure 2-24: Left: Source archive containing the files to extract. Right: Extracted files added to the project.

Figure 2-24. Left: Source archive containing the files to extract. Right: Extracted files added to the project



INFO Pre-made MAC callback functions and other helper files

To simplify the training session, we provide you with four files related to the wireless aspects of the application. These are the files *data_protocol.h*, *data_protocol.c*, *mac_callbacks.c* and *temp_sensor.h*. The pre-made callbacks are implemented in *mac_callbacks.c*:

- `usr_mlme_set_conf()`: Called during configuration when an attribute has been set after calling `wpan_mlme_set_req()`. Depending on the attribute set it will request to set another attribute or signal that the configuration is complete. The implementation of this function can also be found in Application Note AVR2025[14].

[14] http://www.atmel.com/Images/doc5182.pdf

- `usr_mlme_reset_conf()`: Called when a reset has been performed after calling `wpan_mlme_reset_req()`. The header file *temp_sensor.h* defines some constants required by the MAC stack. The files *data_protocol.c* and *data_protocol.h* will be used to send data according to a transfer protocol. This will be covered later.

INFO The MAC stack is very flexible and can be configured according to the application needs. The stack is configured through build switches, which are described in detail in Application Note AVR2025[15].

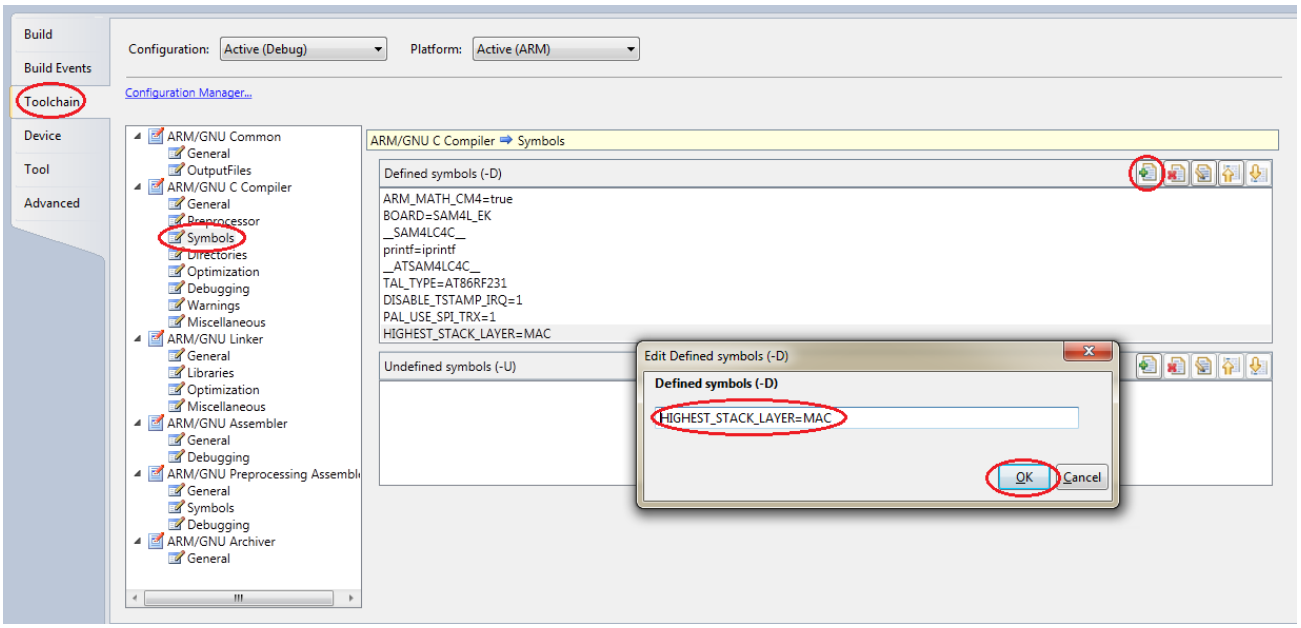
TODO Define the highest stack layer

Before using the wireless MAC service, we must setup our project to be able to use the stack.

We must define a build switch as a symbol to inform the stack which is the highest stack layer. The build switch *HIGHEST_STACK_LAYER* defines the layer that your application is based on, i.e. it determines the API that is used. For our application we want to use the MCL (MAC Core layer). In order to set up *HIGHEST_STACK_LAYER* you can follow these steps (also see Figure 2-25: Define symbol):

- Open the project properties (select `Project > Thermostat Properties ...` or press `Alt + F7`).
- Click on the *Toolchain* tab, select *Arm/GNU C Compiler* and then *Symbols*.
- Click on the icon with the green plus sign on in *Defined symbols*, add *HIGHEST_STACK_LAYER=MAC* and press *OK*. At this point you can close the project properties.

Figure 2-25. Define symbol



TODO Define symbols required by MAC stack

[15] http://www.atmel.com/Images/doc5182.pdf

One of the pre-made files that we included in the project was *temp_sensor.h*. This file defines a number of constants needed by the MAC stack, for example the default communication channel `DEFAULT_CHANNEL`, addresses, ID's, etc. The only thing we need to do is to include the *temp_sensor.h* header file from *main.c*:

```
#include "temp_sensor.h"
```

INFO Board configuration header file

The board configuration file *conf_board.h* is located in the *config* directory and defines a number of symbols related to the hardware setup of the SAM4L-EK. For example, the symbol `CONF_BOARD_BL` allows us to enable the backlight for the LCD display:

```
/* Initialize the LCD Backlight */
#define CONF_BOARD_BL
```

The symbols `CONF_BOARD_SPI` and `CONF_BOARD_SPI_NPCS0` will allow us to use the SPI interface and the chip select lines:

```
/* Initialize the SPI pins for use with the on-board serial flash or with the
 * WIRELESS connector or with the Sensors Xplained extension board. */
#define CONF_BOARD_SPI

/* Initialize the SPI CS for the WIRELESS connector. */
#define CONF_BOARD_SPI_NPCS0
```

The only modification that we need to do in *conf_board.h* is described below.

TODO Configure the board header file according to hardware setup

The symbols `CONF_BOARD_RS485` and `CONF_BOARD_BM_USART` would interfere with the SPI setup required by the RF transceiver. Therefore, we need to comment them out:

```
/* Initialize the USART pins for the RS485 interface */
//#define CONF_BOARD_RS485
```

```
/* Initialize the USART pins that interface with the Board Monitor(BM) */
//#define CONF_BOARD_BM_USART
```

INFO Software timers required in the MAC stack

The MAC service in ASF runs a single hardware timer and emulates several software timers using it. These software timers are used to generate timeouts required by the MAC stack.

TODO Configure software timers required by MAC stack

Open the *conf_common_sw_timer_config.h* configuration file for the software timer. We need to add a constant to let the stack know the total number of software timers. The first thing we need to do is to change the name of the constant `TOTAL_NUMBER_OF_TIMERS` to `TOTAL_NUMBER_OF_SW_TIMERS`. After that we are going to define `TOTAL_NUMBER_OF_SW_TIMERS` as `(TOTAL_NUMBER_OF_TIMERS)`. The constant `TOTAL_NUMBER_OF_TIMERS` is defined in the file *app_config.h* and, therefore, we also need to include that header file from *conf_common_sw_timer_config.h*.

RESULT The resulting configuration in *conf_common_sw_timer.h*:

```
#include "app_config.h"
```

```
#define TOTAL_NUMBER_OF_SW_TIMERS     (TOTAL_NUMBER_OF_TIMERS)
```

WARNING It is easy to miss the fact that you need to add "_SW" between "OF" and "_TIMERS".

TODO Set the number of application timers

Open *app_config.h*, the configuration file for the application, and set `NUMBER_OF_APP_TIMERS`, the number of timers used by the application, to two:

```
#define NUMBER_OF_APP_TIMERS        (2)
```

INFO Pin connections and PAL configuration

The pins used to interface the RZ600-RF231 are defined in *conf_pal.h*, the configuration file for the PAL layer in the MAC stack. The required mapping of these pins on the SAM4L-EK is shown in Figure 2-26: Pin Connections for RZ600-RF231 and SAM4L-EK.

Figure 2-26. Pin Connections for RZ600-RF231 and SAM4L-EK

Pin Name	SAM4LC4CA	SAM4L-EK	RZ600-RF231
RST	PB00	J207.1	1
IRQ	PC02	J207.3	3
SLPTR	PA07	J207.4	4
CS	PA02	J207.5	5
MOSI	PC05	J207.6	6
MISO	PC04	J207.7	7
SCLK	PC06	J207.8	8
Gnd		J207.9	9
Vcc		J207.10	10

INFO Chip Select lines The SAM4L device has four Chip Select lines for SPI. In *conf_board.h* we defined the symbol `CONF_BOARD_SPI_NPCS0`, which enables pin PA02 as Chip Select 0. Therefore, in *conf_pal.h* we can leave the default configuration that sets Chip Select to `0` (instead of defining a pin). This informs the stack that Chip Select 0 is used.

TODO Configure PAL layer

In *conf_pal.h*, scroll down until you find the section that starts with:

```
#if SAM
```

The first thing we need to do is to modify the SPI configuration according to Figure 2-26: Pin Connections for RZ600-RF231 and SAM4L-EK (remember that Chip Select does not need to be modified). In addition, we see that the IRQ line of the RF transceiver is connected to pin PC02. The interrupt handler for this pin is GPIO_8_Handler instead of GPIO_11_Handler. Therefore, we need to replace all references to "GPIO_11" with "GPIO_8".

RESULT The resulting configuration blocks in *conf_pal.h*:

```
#define AT86RFX_SPI                 SPI
#define AT86RFX_RST_PIN             PIN_PB00
#define AT86RFX_IRQ_PIN             PIN_PC02
#define AT86RFX_SLP_PIN             PIN_PA07
#define AT86RFX_SPI_CS              0
#define AT86RFX_SPI_MOSI            PIN_PC05
#define AT86RFX_SPI_MISO            PIN_PC04
#define AT86RFX_SPI_SCK             PIN_PC06
```

```
#define AT86RFX_INTC_INIT()   ioport_set_pin_dir(AT86RFX_IRQ_PIN,\
                              IOPORT_DIR_INPUT);\
                              ioport_set_pin_sense_mode(AT86RFX_IRQ_PIN,\
                              IOPORT_SENSE_RISING);\
                              arch_ioport_pin_to_base(AT86RFX_IRQ_PIN)->\
                              GPIO_IERS = arch_ioport_pin_to_mask(AT86RFX_IRQ_PIN);\
                              arch_ioport_pin_to_base(AT86RFX_IRQ_PIN)->\
                              GPIO_IMR0S = arch_ioport_pin_to_mask(AT86RFX_IRQ_PIN);\
                              NVIC_EnableIRQ(GPIO_8_IRQn)

#define AT86RFX_ISR()         ISR(GPIO_8_Handler)
```

```
#define ENTER_TRX_REGION()        NVIC_DisableIRQ(GPIO_8_IRQn)

/*
 *  This macro restores the transceiver interrupt status
 */
#define LEAVE_TRX_REGION()        NVIC_EnableIRQ(GPIO_8_IRQn)
```

TODO Initialize *RST* and *SLPTR* pins

The pins AT86RFX_RST_PIN and IOPORT_DIR_OUTPUT are used to control the RF transceiver. We need to set them as output pins (IOPORT_DIR_OUTPUT) with high level (IOPORT_PIN_LEVEL_HIGH). You can do this with the functions ioport_set_pin_dir() and ioport_set_pin_level(). These functions receive two parameters: the first one is the pin to modify and the second one is the desired direction or level. The code should be inserted in the main function.

RESULT The following code should be added to the main function before the infinite loop:

```
ioport_set_pin_dir(AT86RFX_RST_PIN, IOPORT_DIR_OUTPUT);
ioport_set_pin_level(AT86RFX_RST_PIN, IOPORT_PIN_LEVEL_HIGH);
ioport_set_pin_dir(AT86RFX_SLP_PIN, IOPORT_DIR_OUTPUT);
ioport_set_pin_level(AT86RFX_SLP_PIN, IOPORT_PIN_LEVEL_HIGH);
```

TODO Initialize delay service

We need to initialize the delay service required by the MAC stack. This can be done by calling the function delay_init() after the other initialization functions.

TODO Initialize software timer

The next step is to initialize the software timer required by the MAC stack. This can be done by calling the function sw_timer_init() after the other initialization functions.

RESULT The following code should be added before the infinite loop in the main function:

```
delay_init();
```

```
sw_timer_init();
```

TODO Initialize all stack layers

To initialize the MAC service, we must add a call to wpan_init(), where WPAN stands for *Wireless Personal Area Network*. This function initializes all the stack layers (PAL, TAL and MAC layers) and returns the status MAC_SUCCESS if the initialization was successful. If the initialization fails, we will simply call a function alert() that will stop further application execution and notify the user (this function will be implemented below).

TODO Implement the function alert()

This function does not take or return any parameters. First, it will turn off all interrupts in the device by calling cpu_irq_disable(). After that it will light a LED on the SAM4L-EK to indicate failure in the wireless stack initialization (use the function ioport_set_pin_level() and the constants LED0_GPIO and

LED0_ACTIVE_LEVEL). Finally, the function will have an infinite loop that does not do anything, thus stopping further execution.

RESULT

The code that initializes the MAC service can be added in the main function after the call to `sw_timer_init()`:

```
if (wpan_init() != MAC_SUCCESS) {
    alert();
}
```

The implementation of the function `alert()` should be included in the main file before the main function

```
static void alert(void)
{
    cpu_irq_disable();
    ioport_set_pin_level(LED0_GPIO, LED0_ACTIVE_LEVEL);

    while (true) {
        /* Do nothing */
    }
}
```

INFO

Notes on the MAC stack API

When the MAC needs to invoke a function in the application, it calls a callback function. Callback functions are prefixed with `usr_` and suffixed with either `_confirm` or `_indication`, for example `usr_mlme_associate_indication`. MAC functions have to be called from the application in order to initiate an action in the communication stack at the MAC level. MAC functions are prefixed with `wpan_` and suffixed with either `_request` or `_response`, for example `wpan_mlme_associate_request()`.

INFO

MAC layer parameters configuration

After the MAC stack is initialized, we need to configure it by calling `wpan_task()` repeatedly. This function will take care of all the MAC layer parameter configuration and further wireless communication. During configuration the MAC service will call several callback functions implemented by the user (remember that you have added *mac_callbacks.c* with pre-made callbacks implementations). When the MAC parameters configuration is done, the flag `radio_ready` is set to `true` at the pre-made callback function `usr_mlme_set_conf()`. The application flow can continue after the MAC layer parameters are configured.

TODO

Configure MAC layer parameters

- The configuration of the stack parameters is done via interrupt callbacks. Therefore, we need to make sure global interrupts are enabled on the device. This can be done by calling `cpu_irq_enable()`.
- Request to reset all the MAC layer parameters to their default value. This can be done by calling the function `wpan_mlme_reset_req()` with the parameter `true`.

- The `radio_ready` variable must be added as a global variable. This is done so we can access it both while waiting for the variable to be set and from the callback function where it will be modified. You can declare this `volatile` variable before the main function, its type is `bool` and it should have an initial value `false`.
- After the reset request, we need to call `wpan_task()` repeatedly until the configuration is done. You can use a `while` loop that repeats while `radio_ready` is `false`.

TODO Notify the user after successful MAC initialization

The LCD display in the SAM4L-EK features a symbol for wireless connections. Therefore, we will show this icon to give a visual confirmation that the MAC stack has been successfully initialized. This can be done by following these steps:

- Open *mac_callbacks.c* and find where the MAC has been correctly initialized (remember the flag `radio_ready`) in the pre-made callback `usr_mlme_set_conf()`.
- Add a wireless icon to the screen by calling the function `c42364a_show_icon()` with the parameter `C42364A_ICON_WLESS`.

RESULT The following global variable should be defined before the main function:

```
volatile bool radio_ready = false;
```

The following code should be added in *mac_callbacks.c* in `usr_mlme_set_conf()` after the flag `radio_ready` is set to `true`:

```
c42364a_show_icon(C42364A_ICON_WLESS);
```

The following code should be added in the main function after the `if` statement that calls `wpan_init()`:

```
cpu_irq_enable();
wpan_mlme_reset_req(true);
while (!radio_ready) {
    wpan_task();
}
```

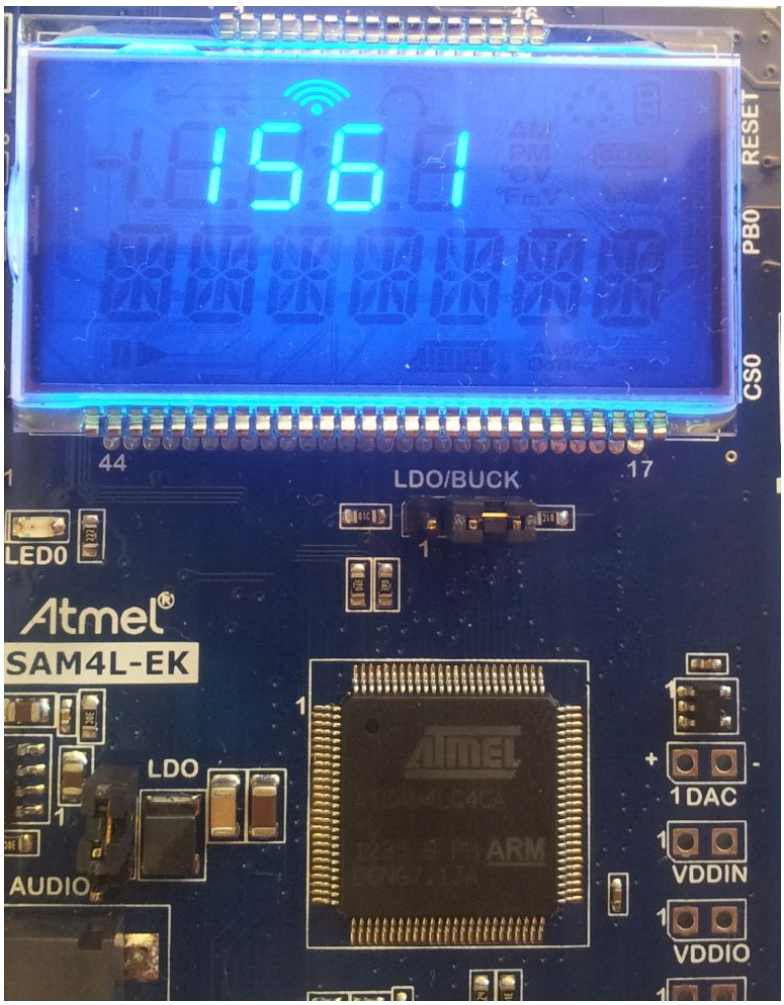
TODO Build and run the application

INFO Third party ASF components

The MAC software package is one of the ASF components of the category Third Party. These are software modules that have been ported to ASF, e.g. an existing module has been modified so that it can be used with the rest of ASF. However, these modules cannot be expected to follow the same standard as native ASF modules. For example, this means that when building a project that includes a 3rd party ASF module, the compiler could raise a number of warnings.

RESULT Your LCD display should be similar to that shown in Figure 2-27: Wireless network correctly initialized, where we can now see the wireless symbol that signals the MAC stack has been initialized correctly.

Figure 2-27. Wireless network correctly initialized



2.4 Sensor Data Transmission

In this section we will enable our smart sensor (thermostat) to communicate with the head-unit (HVAC) through the wireless network. All the smart sensors in the training session will send data and the head-unit will receive and plot the data on the screen.

INFO In the previous section we saw that the function `wpan_task()` runs all the tasks related to the MAC stack. For example, this function ensures that state machines and queues in the MAC stack are served and processed. Therefore, we need to call this function frequently in our application.

TODO Call `wpan_task()` regularly

We will add this function to the infinite while loop of the main function so that `wpan_task()` is called on each run of the loop. You can for example do that before the `if` statement that checks if the state flag `APP_STATE_DISPLAY_RESULT` is set.

```
wpan_task();
```

INFO Data transfer protocol

We will use a protocol for transfering data between your smart sensor (thermostat) and the head-unit (HVAC). The protocol defines packets which consist of a start byte, an address, a bit mask describing which measurements are present in the packet, and the actual data. The address field must be unique for each smart sensor (thermostat) because the head-unit (HVAC) uses this field to determine which smart sensor sent the data. This protocol is already implemented by the head-unit and you just need to configure the protocol and send data in the right format.

TIPS The protocol is supplied in the files *data_protocol.h* and *data_protocol.c*. Remember that you included this files in Add and Setup Wireless MAC Service.

TODO Configure the data transfer protocol

In order to configure the protocol, we need to follow these steps:

- Include the header file *data_protocol.h* from the main file.
- Define the constant `PROTOCOL_ADDRESS` in the main file. This is a unique address for the sensor, whose value will be provided during the training. This ensures that your data is isolated from your neighbor's data by the head-unit (HVAC) since all the nodes are operating with the same channel, PAN ID and short address.
- The data transfer protocol can handle multiple data channels. These data channels need to be defined in an enumeration `protocol_channels`, with one entry for each of the data types we want to send. This would enable the head-unit to distinguish between different types of data sent to it. The protocol has support for up to three different channels but in this training session we will only use one channel `PROTOCOL_LIGHT`.

RESULT The header file should be included from the main file.

```
#include "data_protocol.h"
```

The protocol address constant should be defined before the main function.

```
#define PROTOCOL_ADDRESS 0x32
```

The enumeration `protocol_channels` can be defined before the main function.

```
enum protocol_channels {
    PROTOCOL_LIGHT,
};
```

WARNING Do not use the above value for `PROTOCOL_ADDRESS`, make sure you use the address that is assigned to you by the trainers.

TODO Print protocol address on the LCD display

For debugging purposes, it can be a good idea to show the value of PROTOCOL_ADDRESS on the LCD display. In the main function we will declare a `string_buf` array with the type `uint8_t`. We can store the message to print using the function `snprintf()`. The array can be printed on the LCD display using the function `c42364a_write_alphanum_packet()`.

RESULT The following code should be added in the main function after the initialization of the LCD display:

```
uint8_t string_buf[8];
snprintf(string_buf, 8, "No 0x%2X", PROTOCOL_ADDRESS);
c42364a_write_alphanum_packet(string_buf);
```

TODO Initialize data transfer protocol

Before we can send data we need to initialize the data transfer protocol with the function `protocol_tx_init()`. This function takes two parameters: a pointer to `send_data()`, a function that sends data (we implement it below), and your unique address PROTOCOL_ADDRESS. We can initialize the protocol right before the infinite loop in the main function:

```
protocol_tx_init(send_data, PROTOCOL_ADDRESS);
```

INFO Data transfer protocol portability

The function to send data is provided to the protocol as an argument to the protocol initialization. Therefore, the protocol is independent from the communication channel, i.e. you can also use it in applications that are not using wireless communication.

TODO Implement function `send_data()`

The first thing we need to do is to declare a global variable `dst_addr` with information about the destination address. This variable is a structure of the type `wpan_addr_spec_t` with fields `AddrMode` (we will use `WPAN_ADDRMODE_SHORT`, 16 bit short address mode, defined in *mac_api.h*), `PANid` (we will use `DESTINATION_PAN_ID`, which is defined in *temp_sensor.h*) and `Addr.short_address` (we will use `DESTINATION_SHORT_ADDR`, which is also defined in *temp_sensor.h*).

The `send_data()` function takes as arguments a data buffer (type pointer to `uint8_t`) and the size (type `uint8_t`). This function will call `wpan_mcps_data_req()`, which makes that the data can be sent when calling `wpan_task()` in the infinite loop in the main function. The `wpan_mcps_data_req()` function needs the following arguments:

- Address mode of source address: We will use `WPAN_ADDRMODE_SHORT` (16 bit short address mode)
- A pointer to the destination address specification `dst_addr`.
- Length of data to be sent: This is the `size` parameter of `send_data()`

- Pointer to the data to be sent: This is the data buffer parameter of `send_data()`
- MsduHandle: This is a `static` variable `msduHandle` with type `uint8_t` that identfies the MSDU (MAC service data unit). It will be initialized as zero and incremented on each transmission.
- Transmission options: We will use `WPAN_TXOPT_ACK` (direct acknowledged transmission)

RESULT

The definition of `dst_addr` can be added to the main file before the main function:

```
wpan_addr_spec_t dst_addr = {
   .AddrMode = WPAN_ADDRMODE_SHORT,
   .PANId = DESTINATION_PAN_ID,
   .Addr.short_address = DESTINATION_SHORT_ADDR,
};
```

The definition of `send_data()` can be added to the main file before the main function:

```
static void send_data(
        uint8_t *data,
        uint8_t size)
{
    static uint8_t msduHandle = 0;

    wpan_mcps_data_req(WPAN_ADDRMODE_SHORT, &dst_addr,
            size, data, msduHandle++, WPAN_TXOPT_ACK);
}
```

TODO Add data transmission to the state machine in the application

At this point we have the framework for sending the data and we can add functionality to the application state `APP_STATE_RADIO_TX` that was defined in State Machine. The procedure is similar to the one described in Read and Display Light Sensor Data.

In the infinite loop in the main function there is a block that runs when `APP_STATE_DISPLAY_RESULT` is active. There we can use `protocol_set_channel_data()` to set the data packet that will be sent. This function takes as arguments the data channel number (`PROTOCOL_LIGHT` in this case) and the data buffer (the ADC data buffer `g_adc_sample_data` in this case). After that we can activate the application state `APP_STATE_RADIO_TX`.

```
protocol_set_channel_data(PROTOCOL_LIGHT, &g_adc_sample_data[0]);
set_app_state(APP_STATE_RADIO_TX);
```

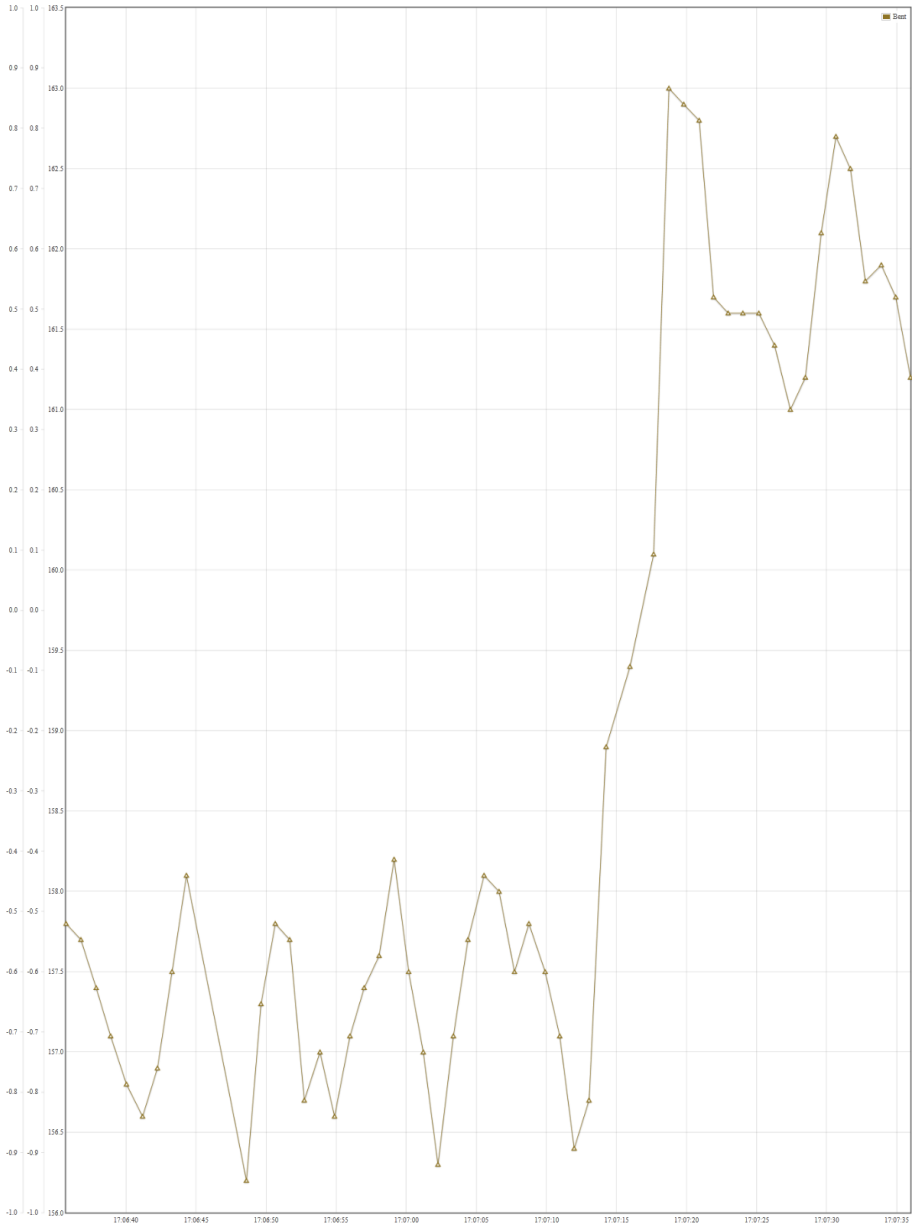
The next step is to add an `if` statement to the infinite loop in the main application to check if the application state `APP_STATE_RADIO_TX` is active. If the state is active, we will first clear the state flag and after that we call the function `protocol_send_packet()`, which will take care of sending of the data.

```
if (is_app_state_set(APP_STATE_RADIO_TX)) {
    clear_app_state(APP_STATE_RADIO_TX);
    protocol_send_packet();
}
```

TODO Build and run the application

RESULT Your smart sensor (thermostat) application is sending data to the head-unit (HVAC) through the wireless network. Therefore, there should be a data graph with your data on the classroom screen, similar to the example shown in Figure 2-28: The head-unit presents the information from all nodes in the classroom.

Figure 2-28. The head-unit presents the information from all nodes in the classroom



2.5 Summary

In this chapter we have shown how to send sensor data from the smart sensor (thermostat) to the head-unit (HVAC) through a wireless network. The head-unit received sensor data and printed a graph for each node in the classroom. The application that we built used the wireless MAC service included in Atmel Software Framework. In addition, data was transferred following an application-independent protocol that allows multiple data channels.

The drivers that have been added are:

- AVR 2025 - IEEE 802.15.4 MAC Configuration

The following RESULT point shows the changes that we have introduced in this chapter. The complete application is presented in Summary.

RESULT The following build switch has been defined in the project properties:

```
HIGHEST_STACK_LAYER=MAC
```

The following has been edited in the configuration file *conf_pal.h:*

```
#define AT86RFX_SPI                 SPI
#define AT86RFX_RST_PIN             PIN_PB00
#define AT86RFX_IRQ_PIN             PIN_PC02
#define AT86RFX_SLP_PIN             PIN_PA07
#define AT86RFX_SPI_CS              0
#define AT86RFX_SPI_MOSI            PIN_PC05
#define AT86RFX_SPI_MISO            PIN_PC04
#define AT86RFX_SPI_SCK             PIN_PC06
```

```
#define AT86RFX_INTC_INIT()   ioport_set_pin_dir(AT86RFX_IRQ_PIN,\
                              IOPORT_DIR_INPUT);\
                              ioport_set_pin_sense_mode(AT86RFX_IRQ_PIN,\
                              IOPORT_SENSE_RISING);\
                              arch_ioport_pin_to_base(AT86RFX_IRQ_PIN)->\
                              GPIO_IERS = arch_ioport_pin_to_mask(AT86RFX_IRQ_PIN);\
                              arch_ioport_pin_to_base(AT86RFX_IRQ_PIN)->\
                              GPIO_IMR0S = arch_ioport_pin_to_mask(AT86RFX_IRQ_PIN);\
                              NVIC_EnableIRQ(GPIO_8_IRQn)

#define AT86RFX_ISR()         ISR(GPIO_8_Handler)
```

```
#define ENTER_TRX_REGION()        NVIC_DisableIRQ(GPIO_8_IRQn)

/*
 *  This macro restores the transceiver interrupt status
 */
#define LEAVE_TRX_REGION()        NVIC_EnableIRQ(GPIO_8_IRQn)
```

The following has been edited in the configuration file *conf_common_sw_timer.h:*

```
#include "app_config.h"
```

```
#define TOTAL_NUMBER_OF_SW_TIMERS     (TOTAL_NUMBER_OF_TIMERS)
```

The following has been edited in the configuration file *app_config.h:*

```
#define NUMBER_OF_APP_TIMERS        (2)
```

The following has been edited in the configuration file *conf_board.h:*

```
/* Initialize the USART pins for the RS485 interface */
//#define CONF_BOARD_RS485
```

```
/* Initialize the LCD Backlight */
#define CONF_BOARD_BL
```

```
/* Initialize the USART pins that interface with the Board Monitor(BM) */
//#define CONF_BOARD_BM_USART
```

```
/* Initialize the SPI pins for use with the on-board serial flash or with the
 * WIRELESS connector or with the Sensors Xplained extension board. */
#define CONF_BOARD_SPI

/* Initialize the SPI CS for the WIRELESS connector. */
#define CONF_BOARD_SPI_NPCS0
```

The following has been added to the main file:

```
#include "temp_sensor.h"
#include "data_protocol.h"
#include "avr2025_mac.h"
#include "pal.h"
```

```
#define PROTOCOL_ADDRESS 0x32


enum protocol_channels {
    PROTOCOL_LIGHT,
};

volatile bool radio_ready = false;
```

```
wpan_addr_spec_t dst_addr = {
   .AddrMode = WPAN_ADDRMODE_SHORT,
   .PANId = DESTINATION_PAN_ID,
   .Addr.short_address = DESTINATION_SHORT_ADDR,
};

static void send_data(
       uint8_t *data,
       uint8_t size)
{
   static uint8_t msduHandle = 0;

   wpan_mcps_data_req(WPAN_ADDRMODE_SHORT, &dst_addr,
           size, data, msduHandle++, WPAN_TXOPT_ACK);
}

static void alert(void)
{
   cpu_irq_disable();
   ioport_set_pin_level(LED0_GPIO, LED0_ACTIVE_LEVEL);

   while (true) {
       /* Do nothing */
   }
}
```

The following has been added to the main function before the infinite loop:

```
ioport_set_pin_dir(AT86RFX_RST_PIN, IOPORT_DIR_OUTPUT);
ioport_set_pin_level(AT86RFX_RST_PIN, IOPORT_PIN_LEVEL_HIGH);
ioport_set_pin_dir(AT86RFX_SLP_PIN, IOPORT_DIR_OUTPUT);
ioport_set_pin_level(AT86RFX_SLP_PIN, IOPORT_PIN_LEVEL_HIGH);

uint8_t string_buf[8];
snprintf(string_buf, 8, "No 0x%2X", PROTOCOL_ADDRESS);
c42364a_write_alphanum_packet(string_buf);

delay_init();

sw_timer_init();

if (wpan_init() != MAC_SUCCESS) {
    alert();
}

cpu_irq_enable();
wpan_mlme_reset_req(true);
while (!radio_ready) {
    wpan_task();
}

protocol_tx_init(send_data, PROTOCOL_ADDRESS);
```

The following has been added to the main function inside the infinite loop:

```
wpan_task();
```

```
if (is_app_state_set(APP_STATE_RADIO_TX)) {
    clear_app_state(APP_STATE_RADIO_TX);
    protocol_send_packet();
}
```

The following was added to the `if` statement that checks the state of `APP_STATE_DISPLAY_RESULT`

```
protocol_set_channel_data(PROTOCOL_LIGHT, &g_adc_sample_data[0]);
set_app_state(APP_STATE_RADIO_TX);
```

3. Develop a Basic Application (OPTIONAL)

TIPS This optional chapter is optional and teaches you how to build the starting application. If you prefer to do this later, you can go directly to Summary.

In this chapter we will develop the application framework. This includes creating a task scheduler and showing data on an LCD display. The overview of this chapter is as follows:

- Basic Initialization
- State Machine
- Task Scheduling
- Displaying Text on an LCD Display
- Summary

3.1 Basic Initialization

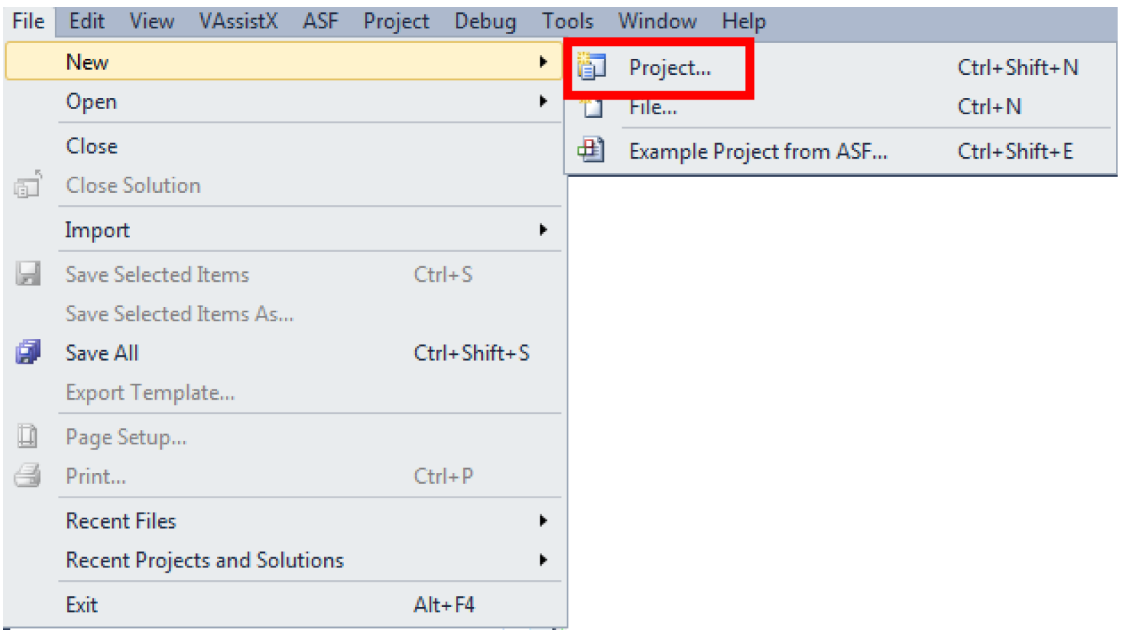
At this point it is assumed that you have installed the training extension, opened Atmel Studio and opened the installed project. In order to create our basic application, we need to add to the existing solution a new project for the SAM4L-EK where we can add drivers and the application code.

WARNING We will *add* this new project to the existing solution. You need to make sure you select "Add to solution" in the next TODO point.

TODO Add a new project for the SAM4L-EK in Atmel Studio 6.1 to your solution

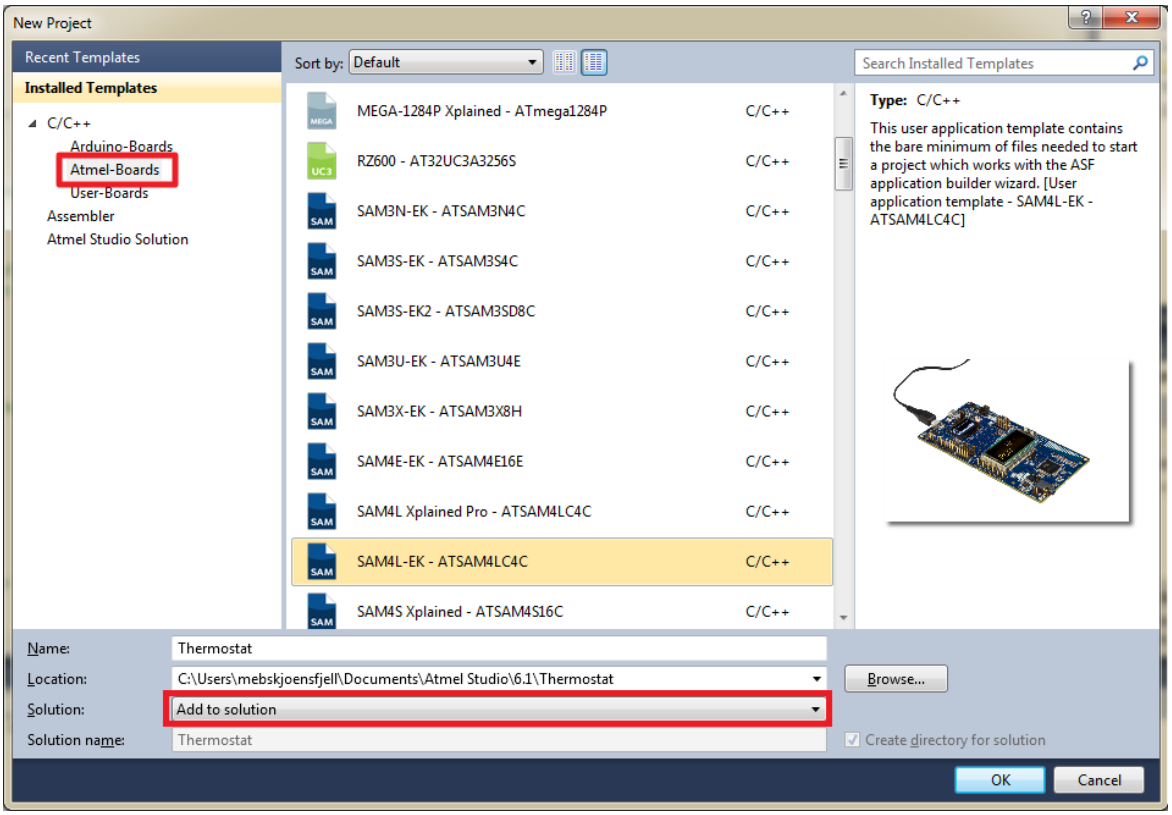
To create a new project, select `File > New > Project...` as shown in Figure 3-1: New Project menu.

Figure 3-1. New Project menu



Select the `C/C++ > Atmel-Boards` template category on the left side of the wizard and then select the SAM4L-EK template item. Give the project a name, for example *Thermostat*. After that make sure you select **Add to solution**. Finally click on *OK* to add the new project to our solution. This is shown in Figure 3-2: SAM4L-EK template selected.

Figure 3-2. SAM4L-EK template selected



TIPS

When there are two projects in a solution, we need to select one of them as the startup-project, i.e. the one that will be compiled and programmed to the device. Check that the "Thermostat" project is shown with bold text in the Solution Explorer. Otherwise right-click it and select *Set as StartUp project*.

INFO

Whenever we create a new project for this board, the autogenerated `main()` (i.e. the main function) includes a call to the function `board_init()`. This function initializes any board specific hardware, e.g. buttons and LEDs. At this point, we need to initialize the clock system, which is described below.

3.1.1 Clock System Initialization and Configuration

In this project, we will use the external crystal running at 12 MHz as the main clock.

TODO

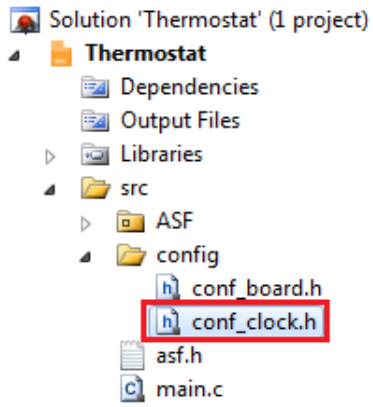
Initialize the clock system

The first step is to initalize the clock system, which can be done by adding a call to `sysclk_init()` into the main function. This function sets up the clock system according to the configuration file *conf_clock.h*. The clock configuration is described in detail below.

INFO Clock Configuration

When a project is created, a configuration file can be found under the folder *src/config*. This file is named *conf_clock.h* and includes defines for all the clock configuration settings of the device for the current project. Figure 3-3: Location of the file conf_clock.h in ASF tree shows this file in the *Solution Explorer*.

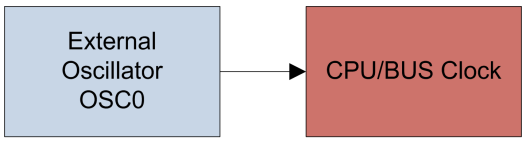
Figure 3-3. Location of the file conf_clock.h in ASF tree



TODO Configure the clock system

We want to configure the external oscillator as main clock as shown in Figure 3-4: Main clock diagram.

Figure 3-4. Main clock diagram



This can be done by modifying existing clock settings in *conf_clock.h*. Modify the existing settings for the constant `CONFIG_SYSCLKC_SOURCE` as described below:

- First, we need to comment out the line that defines `CONFIG_SYSCLKC_SOURCE` as `SYSCLK_SRC_RCSYS`
- After that we can uncomment the line that defines `CONFIG_SYSCLKC_SOURCE` as `SYSCLK_SRC_OSC0`

RESULT The resulting block in *conf_clock.h* is as follows:

```
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_RCSYS
#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_OSC0
```

3.2 State Machine

In this section we will implement a state machine that switches between tasks in the application. We will for example use this to schedule sensor data acquisition in the smart sensor (thermostat).

To switch between states, we will need a framework for the state machine. We will later configure a timer that will handle the setting of the application states based on what needs to be done at which times in the application, such as sampling sensors as well as transmitting data via a wireless link.

TODO Add enumerator with the application task names

Each element will represent one of the tasks in our application: display light sensor value and radio transmit. You can add the following code in `main.c` before the main function.

```
enum app_state {
    APP_STATE_RADIO_TX,
    APP_STATE_DISPLAY_RESULT,
};
```

INFO If at a later point we would like to add a task to our application, we would need to add a new enumerated element.

TODO Add global variable to hold the application state flags

It would be a good idea to allow more than one state to be active at the same time. This is because callbacks associated with various application tasks could be enabled at the same time. Each bit in this variable will be associated with one application task. Therefore, by defining the variable `app_state_flags` as `uint16_t`, we can cater for up to 16 application tasks.

```
volatile uint16_t app_state_flags = 0;
```

Our application may modify this variable from various points (main function and callbacks), so it is best-practice to do so within a *critical section*. In addition, the variable is accessed in a *read-modify-write* instruction (read the application state flag variable, modify by ORing with the state mask, write it back to the variable) and then it is required to modify the variable within a critical section.

The best way to manage the access to the `app_state_flags` variable is through helper functions that safely set, clear and check the application state. This is done in the following TODO points:

TODO Add a function `set_app_state()` that safely sets the application state

The function `set_app_state()` will be `static`, return no value and receive the state to check as a parameter `enum app_state state`. The latter means that we have to include the function definition *after* the enumerator `app_state`. In the function we will:

- Enter the critical section by calling the function `cpu_irq_disable()`, which disables interrupts
- Set the corresponding flag in `app_state_flags` according to `state`
- Leave the critical section by calling the function `cpu_irq_enable()`, which enables interrupts again

The resulting code is given in the next RESULT point. However, before looking at the completed function code, try to build this function yourself!

INFO

Note the use of the qualifier `static` in the function above. In C this qualifier can refer to static storage or static linkage, depending on how it is used:

- Static storage: the most common use is with a variable within a function or block, and in that case the variable would retain its value after the function or block is finished (essentially it is the opposite of auto)
- Static linkage: when a function or variable is declared at the top level (i.e. not within a function) with this qualifier, it will only be visible to the current source file. This can have an impact on code size and optimization

TODO

Add a function `is_app_state_set()` that safely checks if a specified state is set in the global variable

```
static bool is_app_state_set(
        enum app_state state)
{
    bool retval;

    cpu_irq_disable();

    if (app_state_flags & (1 << state)) {
        retval = true;
    } else {
        retval = false;
    }

    cpu_irq_enable();
    return retval;
}
```

TODO

Add a function `clear_app_state()` that safely clears a state flag in the global variable

```
static void clear_app_state(
        enum app_state state)
{
    cpu_irq_disable();

    /* Clear corresponding flag */
    app_state_flags &= ~(1 << state);

    cpu_irq_enable();
}
```

TODO

Add an infinite while loop to *main()*

The next step is to add an infinite loop to the main function. Inside the while loop, we will later add code to check if a particular state is set, and take the appropriate action if it is. Later in this training session we will see how the application states are set when needed.

RESULT At this point we should have the following code before the main function:

```
enum app_state {
    APP_STATE_RADIO_TX,
    APP_STATE_DISPLAY_RESULT,
};

volatile uint16_t app_state_flags = 0;

static bool is_app_state_set(
        enum app_state state)
{
    bool retval;

    cpu_irq_disable();

    if (app_state_flags & (1 << state)) {
        retval = true;
    } else {
        retval = false;
    }

    cpu_irq_enable();
    return retval;
}

static void set_app_state(
        enum app_state state)
{
    cpu_irq_disable();

    /* Set corresponding flag */
    app_state_flags |= (1 << state);

    cpu_irq_enable();
}

static void clear_app_state(
        enum app_state state)
{
    cpu_irq_disable();

    /* Clear corresponding flag */
    app_state_flags &= ~(1 << state);

    cpu_irq_enable();
}
```

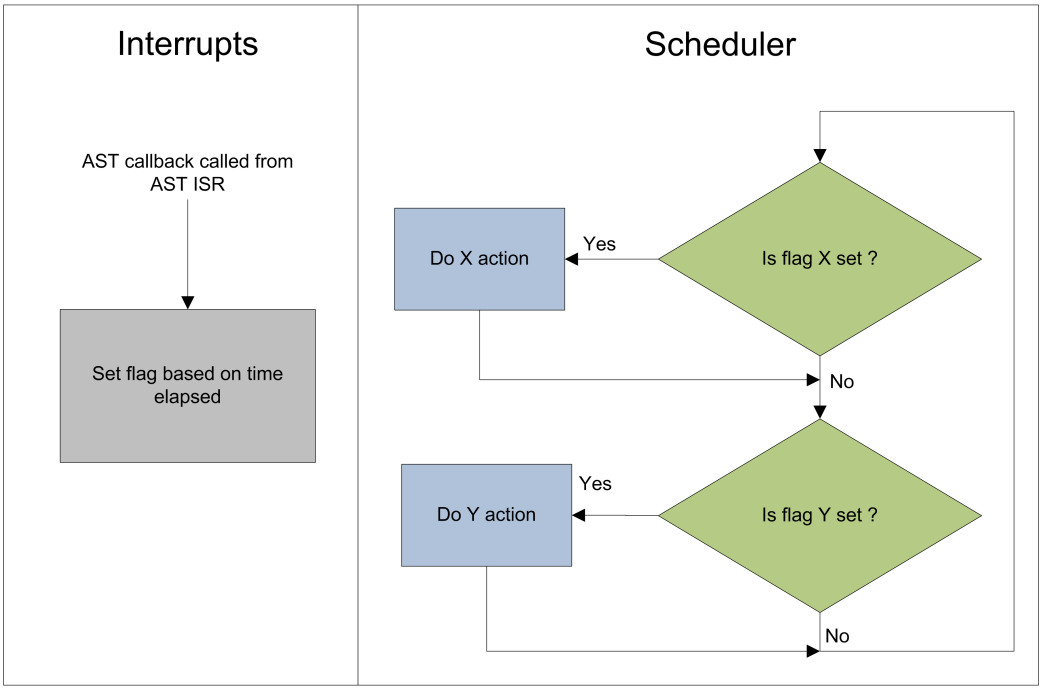
At this point we should have the following code inside the main function:

```
while (true) {
}
```

3.3 Task Scheduling

After building the state machine, our next step is to design a method to switch between tasks. In this section we will use the Asynchronous Timer (AST) to execute a periodic callback that changes the application state. The basic idea is shown in Figure 3-5: Basic Task Scheduling Using an AST.

Figure 3-5. Basic Task Scheduling Using an AST



In the example the AST callback selects one of the states (*X* or *Y*) according to the time elapsed, and the main application executes actions according to the state set in the callback.

In this section we will configure the AST so that we can use it for this simple task scheduling method. Later in the document we will see how we adapt this method to the specific needs of the application. At this point we can advance that a granularity of one second will meet the task switching requirements of our application.

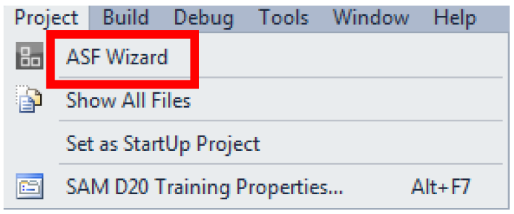
3.3.1 Adding the AST Driver

The first step towards our task scheduler is to add the SAM4L AST driver that is included in ASF.

TODO Add the AST driver using *ASF Wizard*

This can be done by selecting the "Project" menu and then "ASF Wizard" as shown in Figure 3-6: ASF Wizard.

Figure 3-6. ASF Wizard



You should now be presented with a new tab showing the ASF Wizard. On the left hand side you have available modules, and on the right hand side are the modules that are added to your current project.

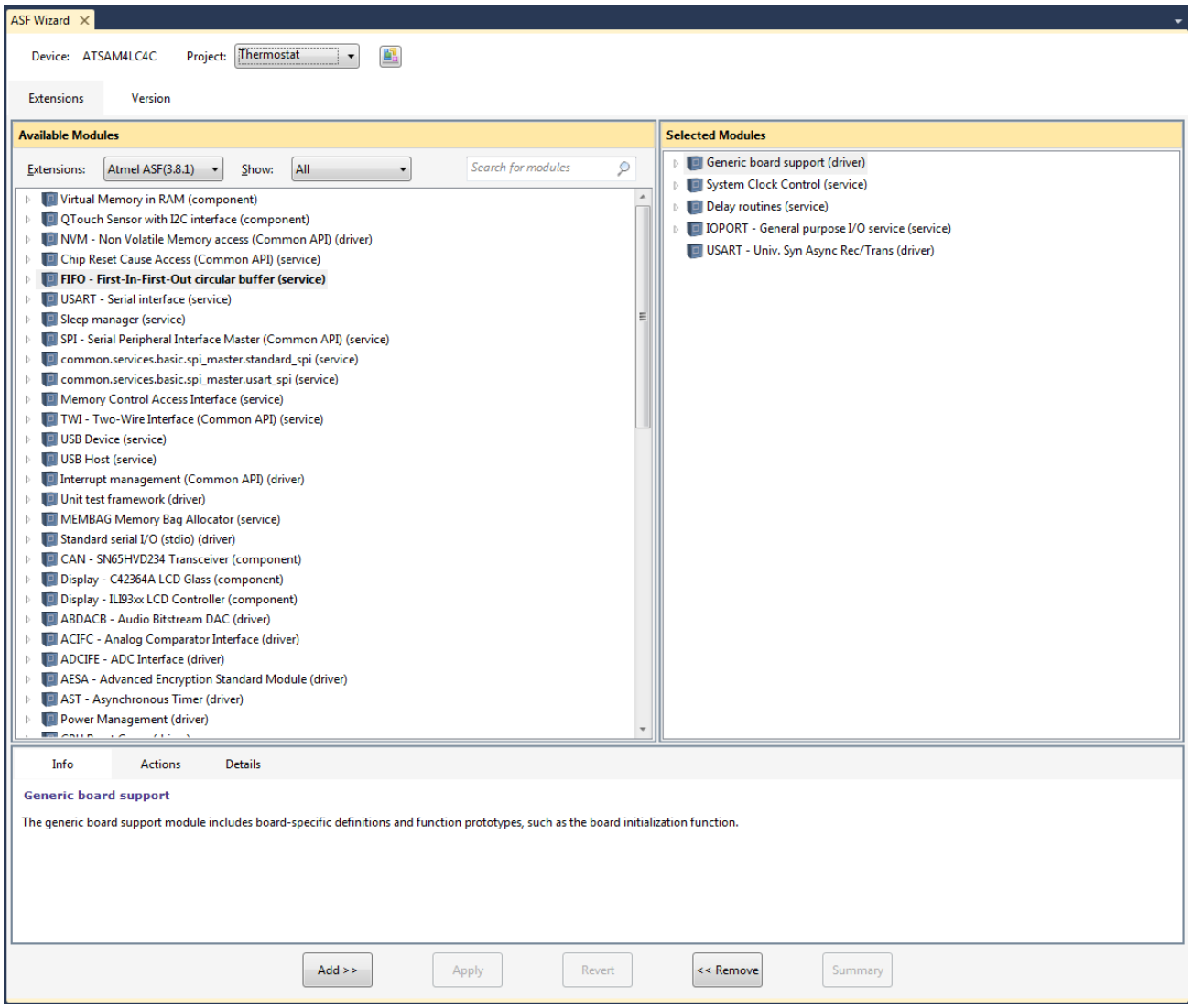
INFO Default drivers

Note that the standard template includes by default the following drivers and services, as shown in :

- Generic board support
- System clock controls
- Delay routines
- IOPORT - General purpose I/O service
- USART - Univ. Syn Async Rec/Trans

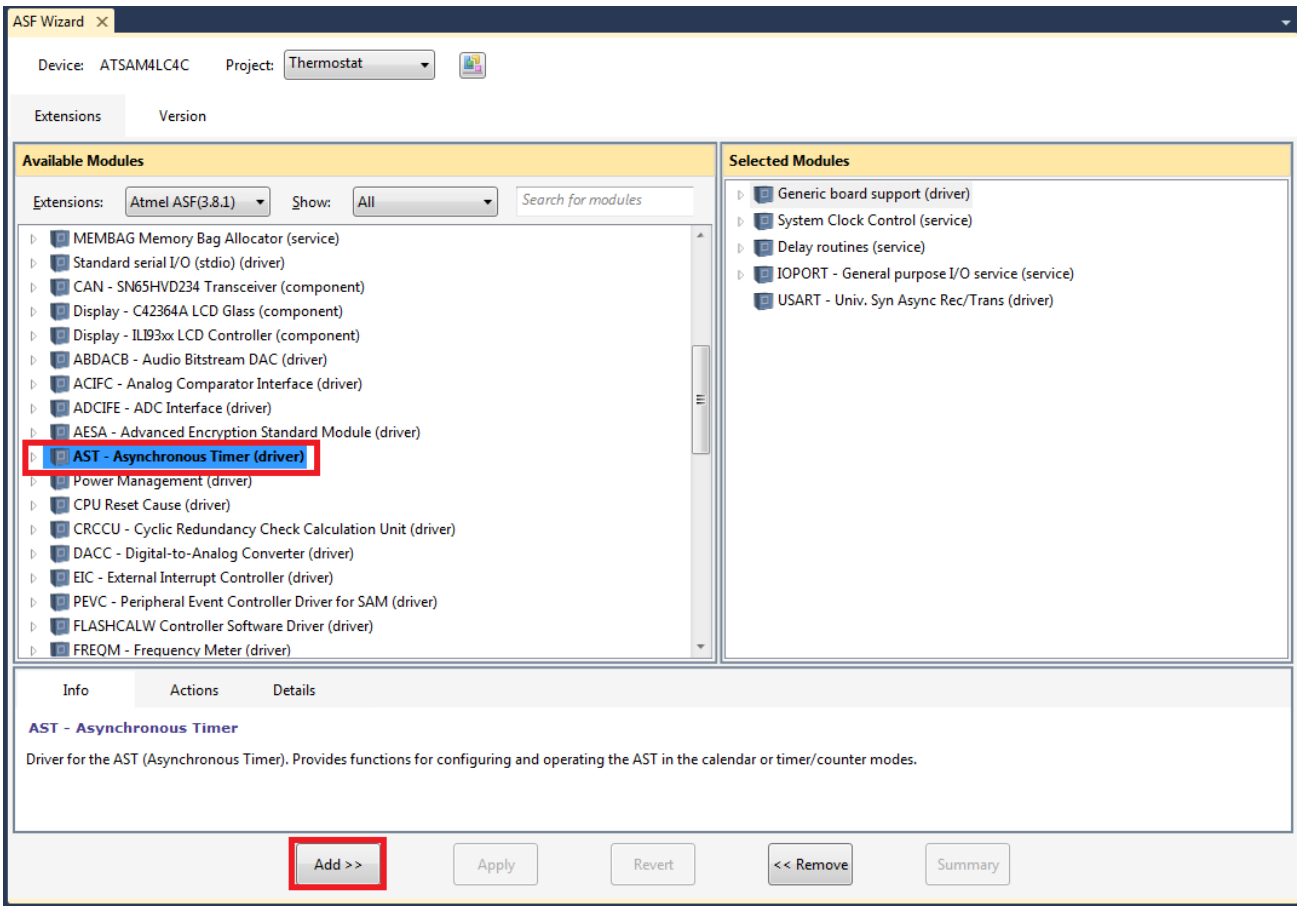
This is shown in Figure 3-7: ASF Wizard with standard board template.

Figure 3-7. ASF Wizard with standard board template



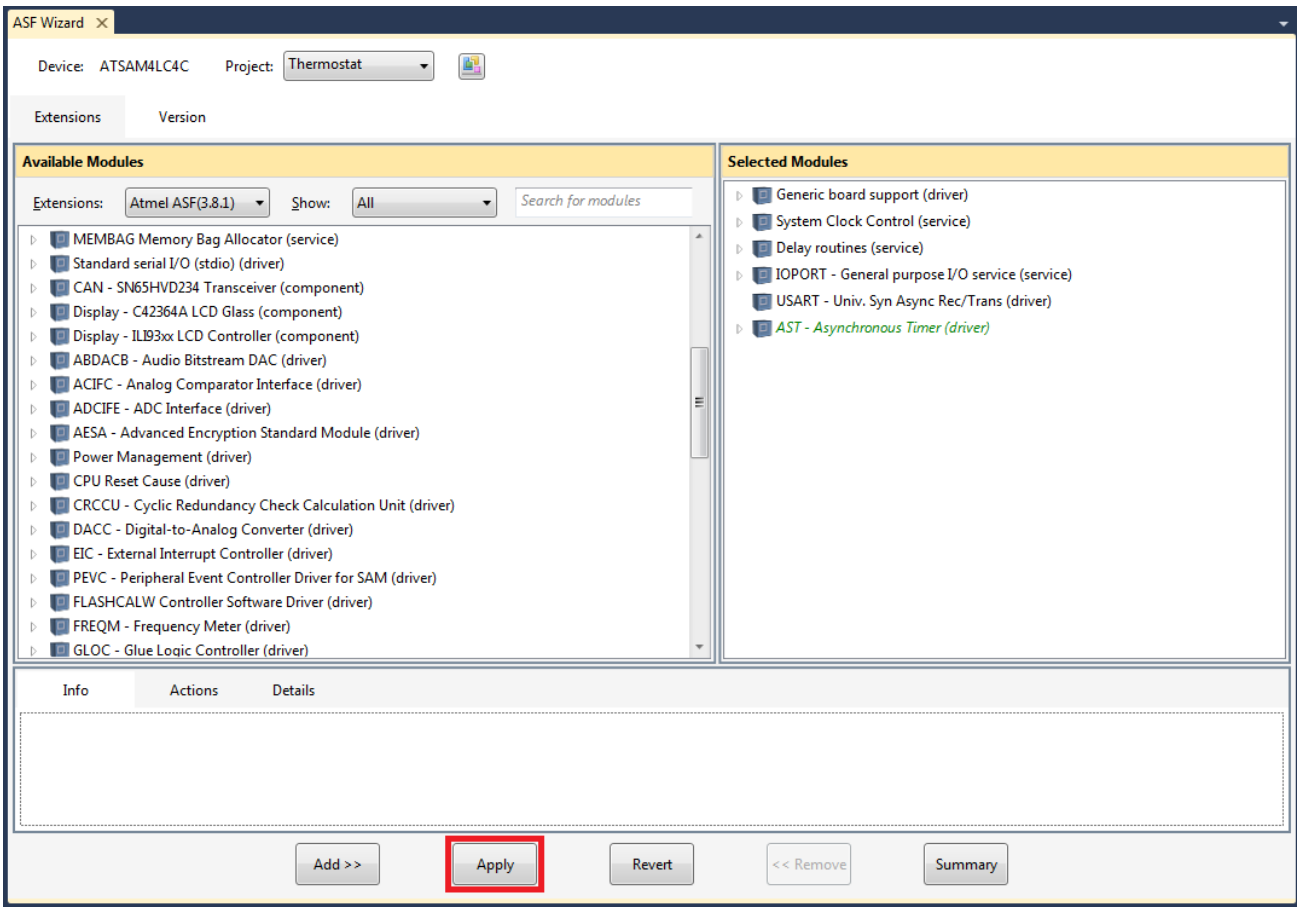
In order to add the AST driver you can select the module from the *Available Modules* on the left side. After that you can press the *Add >>* button to add it to your project. This process is shown in shown in Figure 3-8: AST driver selected.

Figure 3-8. AST driver selected



Finally you need to press the *Apply* button to apply the changes to the project, as shown in Figure 3-9: AST driver added. At this point you can close *ASF Wizard*.

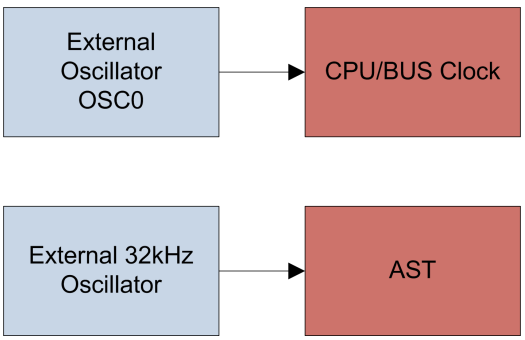
Figure 3-9. AST driver added



3.3.2 Clock Configuration

In our application we will use the 32kHz external crystal for the AST, as shown in the block diagram in Figure 3-10: AST clock diagram. This will improve the power usage and at the same time it offers high accuracy. However, note that in the context of our application AST accuracy is probably not an important feature, e.g. we do not need to read the temperature at specific time intervals, just often enough (e.g. every second).

Figure 3-10. AST clock diagram



In order to obtain the one second time granularity that we want for switching tasks, we will first prescale the AST frequency to obtain approximately 1 kHz clock. After that we will setup an alarm every 1024 cycles, which will give the desired rate of one alarm per second.

TIPS We will write the code that configures the AST clock in Initialize and Enable AST. The alarm configuration is done in Adding the AST Callback.

3.3.3 Initialize and Enable AST

When the driver has been included, the AST needs to be configured and initialized.

TODO Write a function that sets up the AST

We will initialize and enable the AST from a function `ast_setup()`. This function will first enable the AST:

```
ast_enable(AST);
```

We then create a configuration struct used to setup the AST:

```
struct ast_config ast_conf;
```

We edit the configuration struct to use counter mode for the AST.

```
ast_conf.mode = AST_COUNTER_MODE;
```

As discussed in Clock Configuration, we edit the configuration struct to use the 32KHz oscillator with a prescaler value of 4, to get a 1KHz clock.

```
ast_conf.osc_type = AST_OSC_32KHZ;
/* Prescaler that gives an output of 1kHz */
ast_conf.psel = 4;
```

We edit the configuration struct to set the intial value of the timer to zero:

```
ast_conf.counter = 0;
```

Before initializing the AST, we make sure that the OSC32 is enabled and ready:

```
if (!osc_is_ready(OSC_ID_OSC32)) {
    osc_enable(OSC_ID_OSC32);
    osc_wait_ready(OSC_ID_OSC32);
}
```

We then initialize the AST using the configuration struct. We also check whether the initalization is successful, and if not, we turn on `LED0` and stop further application execution.

```
if (!ast_set_config(AST, &ast_conf)) {
    ioport_set_pin_level(LED0_GPIO, LED0_ACTIVE_LEVEL);
    while(true) {
    }
}
```

TODO Call `ast_setup()` from the main function

At this point we have added `ast_setup()` to our application, but we still need to call it from `main()` so that the function is executed.

RESULT The resulting `ast_setup()` code should be included before the main function:

```
static void ast_setup(void)
{
    ast_enable(AST);

    struct ast_config ast_conf;
    ast_conf.mode = AST_COUNTER_MODE;
    ast_conf.osc_type = AST_OSC_32KHZ;
    /* Prescaler that gives an output of 1kHz */
    ast_conf.psel = 4;

    ast_conf.counter = 0;

    /* Enable osc32 oscillator*/
    if (!osc_is_ready(OSC_ID_OSC32)) {
        osc_enable(OSC_ID_OSC32);
        osc_wait_ready(OSC_ID_OSC32);
    }

    /* Initialize the AST */
    if (!ast_set_config(AST, &ast_conf)) {
        ioport_set_pin_level(LED0_GPIO, LED0_ACTIVE_LEVEL);
        while(true) {
        }
    }
}
```

The following code should be included into the main function after the calls to `sysclk_init()` and `board_init()`

```
ast_setup();
```

3.3.4 Adding the AST Callback

The AST is running now, so the next step is to add a callback that can be used for changing the states of the application. The temperature in a room has a very slow dynamic and reading every second will be often enough.

TODO Initialize the interrupt vectors

Since we are using an interrupt, we must initialize the interrupt vectors at the beginning of the main function. This can be done by adding the following code at the beginning of `main()`:

```
irq_initialize_vectors();
```

TODO Add a callback function

For now we just add a simple toggle of the board `LED0` as well as clearing the interrupt flag. This will be useful to check that the AST and its callback are working correctly. You can add the following code before the main function:

```
static void ast_callback(void)
{
    ast_clear_interrupt_flag(AST, AST_INTERRUPT_ALARM);
    ioport_toggle_pin_level(LED0_GPIO);
}
```

INFO This callback function will be called based on the condition we specify when registering and enabling it in the AST driver.

TIPS We will add more code into the AST callback later.

TODO Setup the AST callback

We will setup the AST callback from a function `ast_callback_setup()`. This function will first set an alarm when the counter reaches 1024. This means that the alarm interrupt triggers every second, as we discussed in Clock Configuration.

```
ast_write_alarm0_value(AST, 1024);
```

Then the counter is set to restart at 0 every time the alarm is triggered.

```
ast_enable_counter_clear_on_alarm(AST, 0);
```

The counter is restarted, by setting it to 0, making sure we don't miss the alarm.

```
ast_write_counter_value(AST, 0);
```

The `ast_callback()` function we created is set to be called when the alarm interrupt on the AST is triggered. This will also enable the interrupt for the AST:

```
ast_set_callback(AST, AST_INTERRUPT_ALARM, ast_callback,
        AST_ALARM_IRQn, 0);
```

TODO Call `ast_callback_setup()` from the main function

At this point we have added `ast_callback_setup()` to our application, but we still need to call it from `main()` so that the function is executed.

TIPS You can call `ast_callback_setup()` after the call to `ast_setup()`.

RESULT The resulting code for `ast_callback_setup()` should be included before the main function, after the definition of `ast_callback()`:

```
static void ast_callback_setup(void)
{
    /* Callback to send data over wireless every second */
    ast_write_alarm0_value(AST, 1024);
    ast_enable_counter_clear_on_alarm(AST, 0);
    ast_write_counter_value(AST, 0);
    ast_set_callback(AST, AST_INTERRUPT_ALARM, ast_callback,
            AST_ALARM_IRQn, 0);
}
```

The following code should be included at the top of the main function:

```
irq_initialize_vectors();
```

The following code should be included into the main function after the call to `ast_setup()`:

```
ast_callback_setup();
```

3.3.5 Building and Running the application

In this section we are going to build and run the application that we have coded so far.

TODO Build and run the application

We have two options: We can start a debug session on the board, where we will be able to break and follow the application flow, or we can simply program the compiled code to the controller and execute the application. This is shown in Figure 3-11: Start Without Debugging and Figure 3-12: Start Debugging and Break. We just want to verify our kit, so we will select the "Start Without Debugging".

TIPS If you cannot find the "Start Without Debugging" icon, you can also select `Debug > Start Without Debugging`.

Figure 3-11. Start Without Debugging

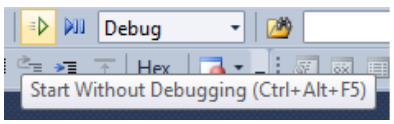
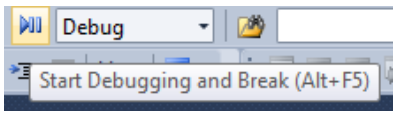


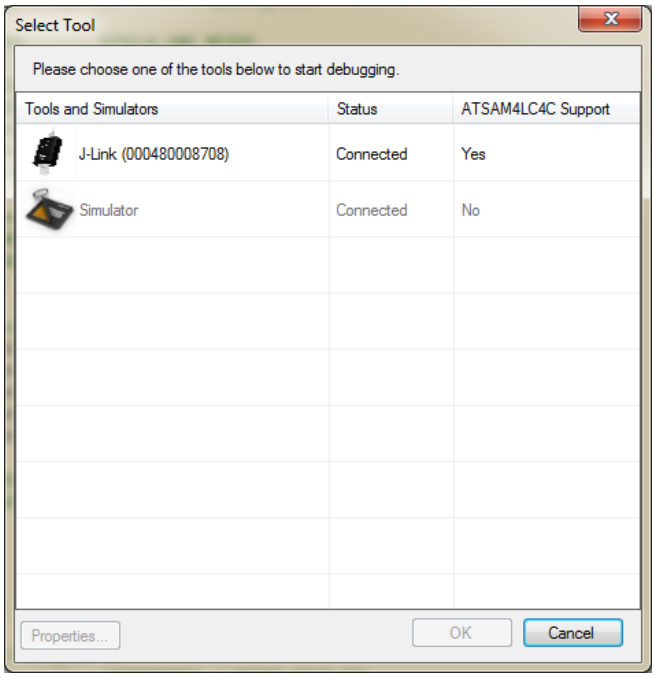
Figure 3-12. Start Debugging and Break



INFO There could be a number of warnings when compiling the application. This is just because the functions that set, clear and check the flag variable `app_state_flags` are defined but not used yet.

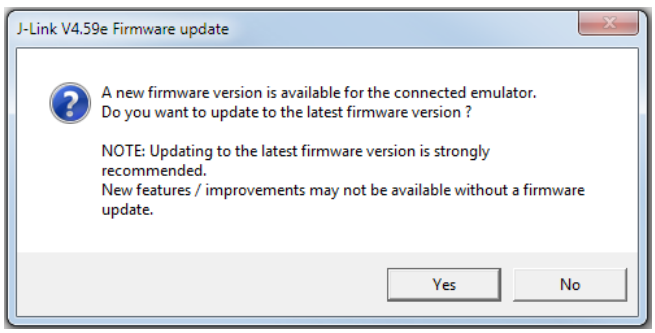
We will now be presented with a tool select dialog that is used to select the correct tool to execute our software on. We will select our SAM4L-EK as shown in Figure 3-13: Select Tool.

Figure 3-13. Select Tool



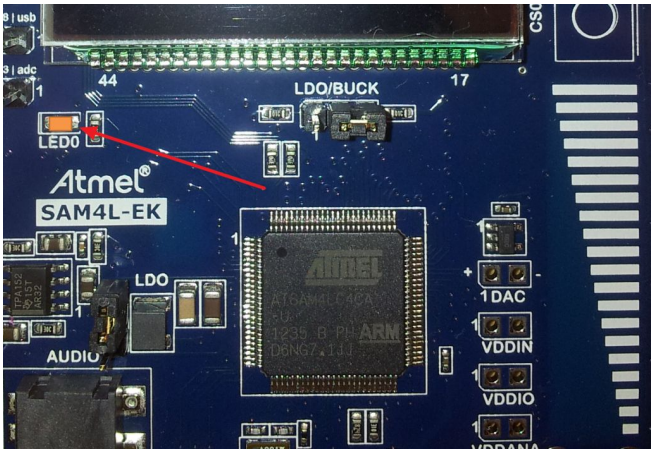
INFO In some cases the firmware of the embedded debugger is out of date, and need to be updated. Atmel Studio will then show a prompt asking to update the firmware. This is shown in Figure 3-14: Firmware update dialog. If this is the case, simply press the "Upgrade" button to upgrade to the latest embedded debugger version.

Figure 3-14. Firmware update dialog



RESULT The board `LED0` should toggle once each second to indicate that the AST is working correctly. This is shown in Figure 3-15: LED0 blinking every second.

Figure 3-15. LED0 blinking every second



3.4 Displaying Text on an LCD Display

In this section we are going to add an LCD driver with string support and print something to the screen. This will be used to display sensor data and the application status.

TODO Add the LCD driver using *ASF Wizard*

We need to add the driver "Display - C42364A LCD Glass". This can be done by selecting the driver, clicking on "Add" and then on "Apply".

TIPS We explained in detail how to add an ASF driver in Adding the AST Driver.

TODO Initialize the LCD display

The LCD can be initialized by calling the function `c42364a_init()`. In addition, we need to enable the LCD backlight by calling `LED_On(LCD_BL)`.

TIPS Call this functions before the infinite loop in the main function.

TODO Print a "Hello world!" string

The first thing we need to do is to put the text in a string variable `scrolling_str`. After that we can print it using the function `c42364a_text_scrolling_start()`. This function receives two parameters, the string variable `scrolling_str` and its size. You should call this function after the screen initialization and before the infinite loop in the main function.

TIPS You can use the function `strlen()` to get the length of a string variable. However, in order to use this function, you need to include *string.h* first (you can do this after *asf.h* is included in the main file).

RESULT The following code has been added on top of the main file:

```
#include <string.h>
```

In addition, the following code has been added before the infinite loop in the main function:

```
c42364a_init();
LED_On(LED_BL);
```

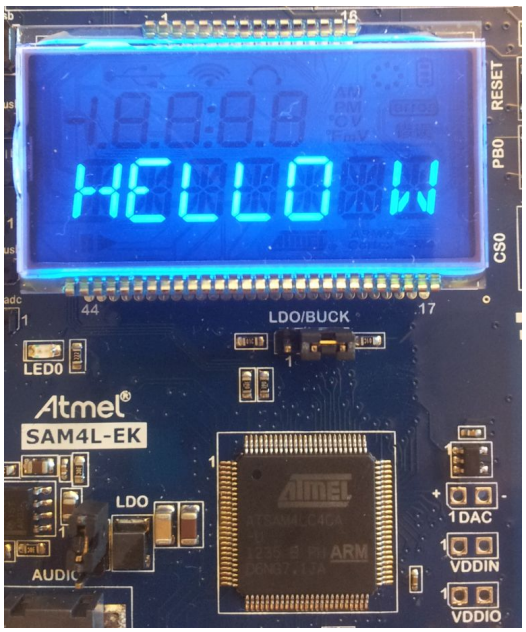
```
uint8_t const scrolling_str[] = "HELLO WORLD!";
c42364a_text_scrolling_start(scrolling_str,
        strlen((char const *) scrolling_str));
```

TODO Build and run the application

The application is ready to be tested. You can click on "Start Without Debugging", as we saw in Building and Running the application.

RESULT At this point you should see the greeting on the LCD display. This is shown in Figure 3-16: Greeting message on LCD display. Note that the `LED0` is still blinking every second.

Figure 3-16. Greeting message on LCD display



3.5 Summary

In this chapter we have developed a basic application that allows us to schedule tasks and print information on the LCD display. We have also learned how to add and configure ASF drivers, and how to run the application in our board.

RESULT The following presents the changes in the application that have been introduced in this chapter.

The ASF drivers that have been added to our project are:

- AST - Asynchronous Timer
- Display - C42364A LCD Glass

The following configuration settings have been edited in *conf_clock.h*:

```
//#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_RCSYS
#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_OSC0
```

The following file has been included in *main.c*:

```
#include <string.h>
```

The following code has been added before the main function:

```
enum app_state {
    APP_STATE_RADIO_TX,
    APP_STATE_DISPLAY_RESULT,
};

volatile uint16_t app_state_flags = 0;

static bool is_app_state_set(
        enum app_state state)
{
    bool retval;

    cpu_irq_disable();

    if (app_state_flags & (1 << state)) {
        retval = true;
    } else {
        retval = false;
    }

    cpu_irq_enable();
    return retval;
}

static void set_app_state(
        enum app_state state)
{
    cpu_irq_disable();

    /* Set corresponding flag */
    app_state_flags |= (1 << state);

    cpu_irq_enable();
}

static void clear_app_state(
        enum app_state state)
{
    cpu_irq_disable();

    /* Clear corresponding flag */
    app_state_flags &= ~(1 << state);

    cpu_irq_enable();
```

```
}
```

```
static void ast_setup(void)
{
    ast_enable(AST);

    struct ast_config ast_conf;
    ast_conf.mode = AST_COUNTER_MODE;
    ast_conf.osc_type = AST_OSC_32KHZ;
    /* Prescaler that gives an output of 1kHz */
    ast_conf.psel = 4;

    ast_conf.counter = 0;

    /* Enable osc32 oscillator*/
    if (!osc_is_ready(OSC_ID_OSC32)) {
        osc_enable(OSC_ID_OSC32);
        osc_wait_ready(OSC_ID_OSC32);
    }

    /* Initialize the AST */
    if (!ast_set_config(AST, &ast_conf)) {
        ioport_set_pin_level(LED0_GPIO, LED0_ACTIVE_LEVEL);
        while(true) {
        }
    }
}
```

```
static void ast_callback(void)
{
    ast_clear_interrupt_flag(AST, AST_INTERRUPT_ALARM);
    ioport_toggle_pin_level(LED0_GPIO);
}
```

```
static void ast_callback_setup(void)
{
    /* Callback to send data over wireless every second */
    ast_write_alarm0_value(AST, 1024);
    ast_enable_counter_clear_on_alarm(AST, 0);
    ast_write_counter_value(AST, 0);
    ast_set_callback(AST, AST_INTERRUPT_ALARM, ast_callback,
            AST_ALARM_IRQn, 0);
}
```

The resulting main function:

```
int main (void)
{
```

```
irq_initialize_vectors();
sysclk_init();
board_init();
```

```
ast_setup();
ast_callback_setup();
```

```
c42364a_init();
LED_On(LCD_BL);
```

```
uint8_t const scrolling_str[] = "HELLO WORLD!";
c42364a_text_scrolling_start(scrolling_str,
        strlen((char const *) scrolling_str));
```

```
while (true) {
}
```

```
}
```

4. Light Sensor Acquisition (OPTIONAL)

TIPS This optional chapter is optional and teaches you how to build the starting application. If you prefer to do this later, you can go directly to Summary.

In this chapter we measure the analog value produced by the light sensor that is available on the SAM4L-EK board and display it on the LCD display.

The overview of this chapter is as follows:

- Introduction to the Light Sensor
- appdoc_chap_light_acquire
- Add and Configure ADCIFE driver
- Read and Display Light Sensor Data
- Summary

4.1 Introduction to the Light Sensor

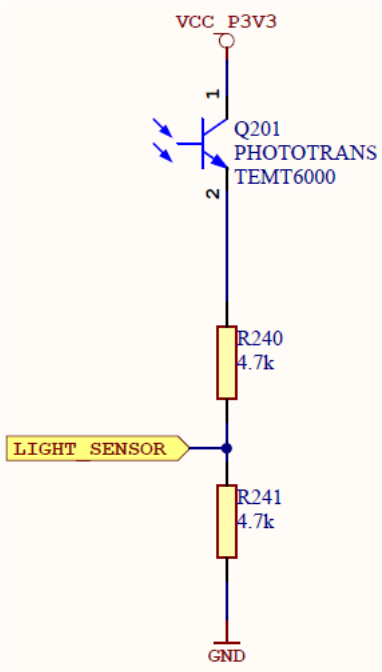
As we discussed in Introduction, for simplicity we use a light sensor instead of a temperature sensor to build or smart sensor (thermostat) demo. The light sensor that is available on the SAM4L-EK is the TEMT6000 ambient light sensor from Vishay. (Datasheet[1]). The sensor is a photo transistor, where the collector/emitter voltage will vary according to the light applied to the sensor. The sensor is sensitive to light in the visible spectra, much like the human eye.

On the SAM4L-EK the TEMT6000 is connected as shown in Figure 4-1: Light sensor schematic, where the ADC is connected below the light sensor, measuring the voltage across the sensor. A low ADC measurement signifies a small amount of light hitting the sensor, while a high ADC reading will signify a larger amount of light hitting the sensor. Note that the voltage will never quite reach a value of zero, as there will then be a saturation voltage across the collector emitter of typically 0.1 V (from TEMT6000 datasheet).

When measuring the voltage across the light sensor we will be using the ADC in a 12-bit, single-ended mode.

[1] http://www.vishay.com/product?docid=81579

Figure 4-1. Light sensor schematic



4.2 Add and Configure ADCIFE driver

In this section we will add the software driver for the ADCIF and configure it to sample the light sensor analog value. When our given number of samples has been sampled, a callback function specified in the application will be called, and signals the application that data is ready.

TODO Add ADC Interface driver using the ASF wizard.

We need to add the driver "ADCIFE - ADC Interface (driver)". This can be done by selecting the driver, clicking on "Add" and then on "Apply".

TIPS We explained in detail how to add an ASF driver in Adding the AST Driver.

When the driver has been included, the ADC needs to be initialized and configured for our application.

INFO The code that we are using to initialize the ADC is based on the ADCIFE driver Quick Start Guide and the ADC Interface example in ASF. This chapter is selfcontained, but if you want to have a look at them later:

- The Quick Start Guide can be found as follows: double-clik on the API documentation (ASF `Explorer > ADCIFE - ADC Interface > API Documentation`, a webpage will open on your browser) and follow the link "Quickstart guide for SAM ADCIFE driver" on top of the page. This guide is only available online, but you can also find it in *adcife.h* (you can access this from ASF `Explorer > ADCIFE - ADC Interface`.
- The example can be opened as follows: select `File > New > Example Project..` and choose "ADC Interface Example - SAM4L-EK".

TODO Add to the application a buffer to store sensor data

We will start by declaring a buffer where we can save the data acquired by the ADC. Consider the following points:

- Each sample has 12 bits, so we need to declare a buffer with `uint16_t` type.
- The buffer variable will be an array called `g_adc_sample_data`.
- This variable has to be global so that it can be accessed from the main function and the interrupts.
- The size of the array will be defined in a constant `APP_ADC_SAMPLES`.
- In our application we will use a buffer of size 1, i.e. define `APP_ADC_SAMPLES` with that value.

INFO Using an array instead of a one-dimensional variable is more general and allows us to easily aqcuire and process more samples if we need to do this later.

RESULT The code that declares the data buffer should be added at the top of the main file:

```
#define APP_ADC_SAMPLES    1
uint16_t g_adc_sample_data[APP_ADC_SAMPLES];
```

TODO Add ADC setup code and callback

The `adc_setup()` function will initialize and enable the ADC, and register the callback function `adcife_read_conv_result()`. The callback function is called each time a new ADC sample is ready and it will just add the sample to the buffer g_adc_sample_data that we have declared.

TIPS During the design of the application we studied the Quick Start Guide and the example provided in ASF. We found that the provided code fulfilled our requirements and then we used it for our application. For simplicity, we give you the code in the section below.

RESULT The definitions of `adc_setup()` and `adcife_read_conv_result()` should be included before the main function, e.g. after the definition of `g_adc_sample_data:`

```
static void adcife_read_conv_result(void)
{
    // Check the ADC conversion status
    if ((adc_get_status(&g_adc_inst) & ADCIFE_SR_SEOC) == ADCIFE_SR_SEOC){
        g_adc_sample_data[0] = adc_get_last_conv_value(&g_adc_inst);
        adc_clear_status(&g_adc_inst, ADCIFE_SCR_SEOC);
    }
}
```

```
void adc_setup(void)
{
    adc_init(&g_adc_inst, ADCIFE, &adc_cfg);
    adc_enable(&g_adc_inst);
    adc_ch_set_config(&g_adc_inst, &adc_ch_cfg);
    adc_set_callback(&g_adc_inst, ADC_SEQ_SEOC, adcife_read_conv_result,
            ADCIFE_IRQn, 1);
}
```

INFO Note that we have avoided to do any heavy processing inside the callback: the sample is simply added to the buffer. Given that the callback runs in an interrupt context, we should exit as soon as possible and return to the application code (the main execution flow). This is considered best-practice for embedded applications.

TODO Add software instance and configuration structures

Note that the code for `adc_setup()` and `adcife_read_conv_result()` require a software instance `g_adc_inst` (a data structure used by the driver to store the state of the ADC) and configuration stuctures `adc_cfg` and `adc_ch_cfg` (data structures that save configuration settings for the ADC).

TIPS The Quick Start guide is missing this information, so we need to look for the information somewhere else. In this case we find that the example we mentioned above (i.e. "ADC Interface Example - SAM4L-EK") gives us this information. For simplicity, we include the required steps below:

For the software instance, we just need to declare a variable `g_adc_inst` with the type *struct adc_dev_inst*. It is required that `g_adc_inst` is available globally because it will be used both in the initalization and in the AST callback for triggering new conversions. Therefore, the declaration should be on top of the main file, before the declarations of `adc_setup()` and `adcife_read_conv_result()`.

```
struct adc_dev_inst g_adc_inst;
```

In order to create the configuration structs, we can copy the definitions of `adc_cfg`, `adc_seq_cfg` and `adc_ch_cfg`, which are found in the ASF example inside the function `start_adc()`. The only differences with respect to the example are as follows:

- We need to edit the positive mux selection (`muxpos`) in `adc_seq_cfg` to `ADC_MUXPOS_6`. This is because the light sensor is connected to PB05, which is the sixth channel for the ADC.

```
.muxpos = ADC_MUXPOS_6,
```

- We also need to edit the internal voltage sources selection (`internal`) in `adc_seq_cfg` to `ADC_INTERNAL_2`. This is because we want to read from the pin 6 of the ADC and not from the DAC internal voltage as in the ASF example. In order to get the information, we actually had to check the datasheet[2] (ADCIFE chapter, page 1009)

```
.internal = ADC_INTERNAL_2,
```

RESULT The resulting configuration structures should be defined on top of the function `adc_setup()`

[2] http://www.atmel.com/Images/Atmel-42023-ARM-Microcontroller-ATSAM4L-Low-Power-LCD_Datasheet.pdf

```
struct adc_config adc_cfg = {
    /* System clock division factor is 16 */
    .prescal = ADC_PRESCAL_DIV16,
    /* The APB clock is used */
    .clksel = ADC_CLKSEL_APBCLK,
    /* Max speed is 150K */
    .speed = ADC_SPEED_150K,
    /* ADC Reference voltage is 0.625*VCC */
    .refsel = ADC_REFSEL_1,
    /* Enables the Startup time */
    .start_up = CONFIG_ADC_STARTUP
};
struct adc_seq_config adc_seq_cfg = {
    /* Select Vref for shift cycle */
    .zoomrange = ADC_ZOOMRANGE_0,
    /* Pad Ground */
    .muxneg = ADC_MUXNEG_1,
    /* Scaled Vcc, Vcc/10 */
    .muxpos = ADC_MUXPOS_6,
    /* Enables the internal voltage sources */
    .internal = ADC_INTERNAL_2,
    /* Disables the ADC gain error reduction */
    .gcomp = ADC_GCOMP_DIS,
    /* Disables the HWLA mode */
    .hwla = ADC_HWLA_DIS,
    /* 12-bits resolution */
    .res = ADC_RES_12_BIT,
    /* Enables the single-ended mode */
    .bipolar = ADC_BIPOLAR_SINGLEENDED
};

struct adc_ch_config adc_ch_cfg = {
    .seq_cfg = &adc_seq_cfg,
    /* Internal Timer Max Counter */
    .internal_timer_max_count = 60,
    /* Window monitor mode is off */
    .window_mode = 0,
    .low_threshold = 0,
    .high_threshold = 0,
};
```

Setup the ADC from the application main function.

Once we have added to the application the functions `adc_setup()` and `adcife_read_conv_result()`, we should call the `adc_setup()` from the main function so that the initialization is run.

RESULT The following code should be included after the code that sets up the LCD display:

```
adc_setup();
```

4.3 Read and Display Light Sensor Data

After completing the setup and initialization of the ADC, in this section we will read light sensor data and print it on the LCD display. This will enable us to check and interact with the system we have built so far: if we cover the light sensor, we should see changes in the displayed value.

TODO Start an ADC conversion from the AST callback function

We will start the ADC conversion from the AST callback function `ast_callback()`, which gets executed every second according to the configuration in Task Scheduling. We can start an ADC conversion just by calling the function `adc_start_software_conversion()`, which takes a pointer to the ADC software instance as a parameter.

TODO Signal the main application when ADC data is to be displayed

We want to show sensor data on the LCD display every time a new ADC sample is ready. For this we can use the state machine framework we set up in Develop a Basic Application (OPTIONAL). The first thing we will do is to set the state flag for the state `APP_STATE_DISPLAY_RESULT` once a new data sample is ready to be shown on the LCD display. Remember that you can set the state using the function `set_app_state()`. The ADC callback function is executed after the ADC sample is available, so we can just set the flag from `adcife_read_conv_result()`. You can do this inside inside the `if` statement, at the end.

TODO Display data when the flag is set

The following step is to check whether `APP_STATE_DISPLAY_RESULT` is set. This can be done with an `if` statement in the infinite loop in the main function. Remember that you can check if a state is active using the function `is_app_state_set()`. If the flag is active, the first thing to do is to clear it so that we avoid printing the ADC value again in the next iteration of the infinite loop in the main function. Remember that you can clear a state using the function `clear_app_state()`. After that we can print the data sample using the function `c42364a_show_numeric_dec()`, which takes as single parameter the data to print (i.e. our data buffer).

TIPS In order to avoid having the "HELLO WORLD!" string scrolling on the LCD display, we can comment out the call to `c42364a_text_scrolling_start()` before the infinite loop.

RESULT The following code should be included at the end of `ast_callback()` to trigger new conversion:

```
adc_start_software_conversion(&g_adc_inst);
```

The following code should be included at the end of the `if` statement in `adcife_read_conv_result()` to set the state flag:

```
set_app_state(APP_STATE_DISPLAY_RESULT);
```

This block should be included inside the infinite loop to print data on the LCD display:

```
if (is_app_state_set(APP_STATE_DISPLAY_RESULT)) {
    clear_app_state(APP_STATE_DISPLAY_RESULT);
    c42364a_show_numeric_dec(g_adc_sample_data[0]);
}
```

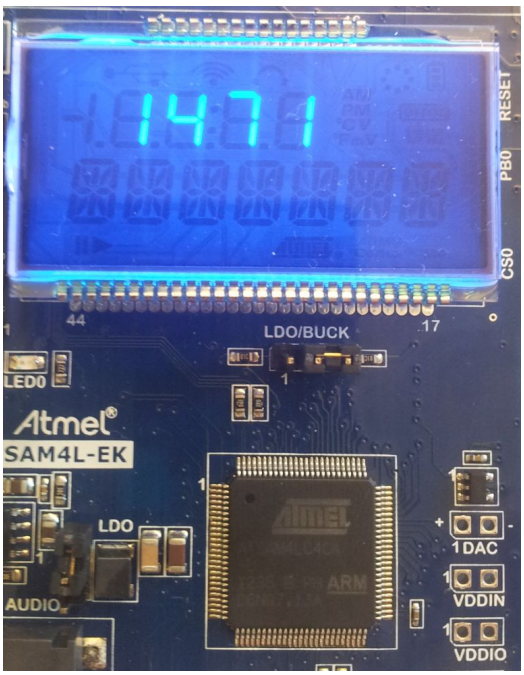
If we want to avoid having the "HELLO WORLD!" message, we have to comment out the following code before the infinite loop:

```
//uint8_t const scrolling_str[] = "HELLO WORLD!";
//c42364a_text_scrolling_start(scrolling_str,
//      strlen((char const *) scrolling_str));
```

TODO Build and run the application

RESULT At this point you should see the value read from the light sensor updated on the display every second,as shown in Figure 4-2: Sample data on the LCD display. If you cover the light sensor (located on the top left corner of the board) with your finger, the value shown in the LCD display should change.

Figure 4-2. Sample data on the LCD display



4.4 Summary

In this chapter we have learned how to add the ADCIFE driver from ASF, how to configure the ADC to sample light sensor data and how to access the sampled data. In addition, we used the LCD display to print the sensor data read by the ADC. This allowed us to interact with our application and check the system.

The ASF drivers that have been added are:

- ADCIFE

RESULT At this point we should have added the following code to *main.c*:

```
#define APP_ADC_SAMPLES    1
uint16_t g_adc_sample_data[APP_ADC_SAMPLES];
```

```
struct adc_dev_inst g_adc_inst;
```

```
static void adcife_read_conv_result(void)
{
    // Check the ADC conversion status
    if ((adc_get_status(&g_adc_inst) & ADCIFE_SR_SEOC) == ADCIFE_SR_SEOC){
        g_adc_sample_data[0] = adc_get_last_conv_value(&g_adc_inst);
        adc_clear_status(&g_adc_inst, ADCIFE_SCR_SEOC);
        set_app_state(APP_STATE_DISPLAY_RESULT);
    }
}
```

```
static void adc_setup(void)
{
    struct adc_config adc_cfg = {
        /* System clock division factor is 16 */
        .prescal = ADC_PRESCAL_DIV16,
        /* The APB clock is used */
        .clksel = ADC_CLKSEL_APBCLK,
        /* Max speed is 150K */
        .speed = ADC_SPEED_150K,
        /* ADC Reference voltage is 0.625*VCC */
        .refsel = ADC_REFSEL_1,
        /* Enables the Startup time */
        .start_up = CONFIG_ADC_STARTUP
    };
    struct adc_seq_config adc_seq_cfg = {
        /* Select Vref for shift cycle */
        .zoomrange = ADC_ZOOMRANGE_0,
        /* Pad Ground */
        .muxneg = ADC_MUXNEG_1,
        /* Scaled Vcc, Vcc/10 */
        .muxpos = ADC_MUXPOS_6,
        /* Enables the internal voltage sources */
        .internal = ADC_INTERNAL_2,
        /* Disables the ADC gain error reduction */
        .gcomp = ADC_GCOMP_DIS,
        /* Disables the HWLA mode */
        .hwla = ADC_HWLA_DIS,
        /* 12-bits resolution */
        .res = ADC_RES_12_BIT,
        /* Enables the single-ended mode */
        .bipolar = ADC_BIPOLAR_SINGLEENDED
    };

    struct adc_ch_config adc_ch_cfg = {
        .seq_cfg = &adc_seq_cfg,
        /* Internal Timer Max Counter */
        .internal_timer_max_count = 60,
        /* Window monitor mode is off */
        .window_mode = 0,
        .low_threshold = 0,
        .high_threshold = 0,
    };

    adc_init(&g_adc_inst, ADCIFE, &adc_cfg);
    adc_enable(&g_adc_inst);
    adc_ch_set_config(&g_adc_inst, &adc_ch_cfg);
    adc_set_callback(&g_adc_inst, ADC_SEQ_SEOC, adcife_read_conv_result,
```

```
    ADCIFE_IRQn, 1);
}
```

Inside the `ast_callback()` function:

```
adc_start_software_conversion(&g_adc_inst);
```

Inside the `adcife_read_conv_result()` function:

```
set_app_state(APP_STATE_DISPLAY_RESULT);
```

Inside `main()`:

```
adc_setup();
```

```
//uint8_t const scrolling_str[] = "HELLO WORLD!";
//c42364a_text_scrolling_start(scrolling_str,
//      strlen((char const *) scrolling_str));
```

Inside the infinite while loop of the application:

```
if (is_app_state_set(APP_STATE_DISPLAY_RESULT)) {
    clear_app_state(APP_STATE_DISPLAY_RESULT);
    c42364a_show_numeric_dec(g_adc_sample_data[0]);
}
```

5. Summary

In this hands-on session we have built a smart sensor (thermostat) that communicates with a head-unit (HVAC) through a wireless network. The smart sensor reads sensor data with the ADC, shows the information on the LCD display and sends it to a head-unit. The provided head-unit receives data from all the smart sensors in the classroom and plots one graph for each node.

Your smart sensor has been built with an SAM4L-EK and the RF transceiver RZ600-RF231. The application has been developed with Atmel Studio and Atmel Software Framework, which includes a wireless MAC stack. The design process that we have covered in this training session is general and applies to any device supported by the MAC stack in ASF.

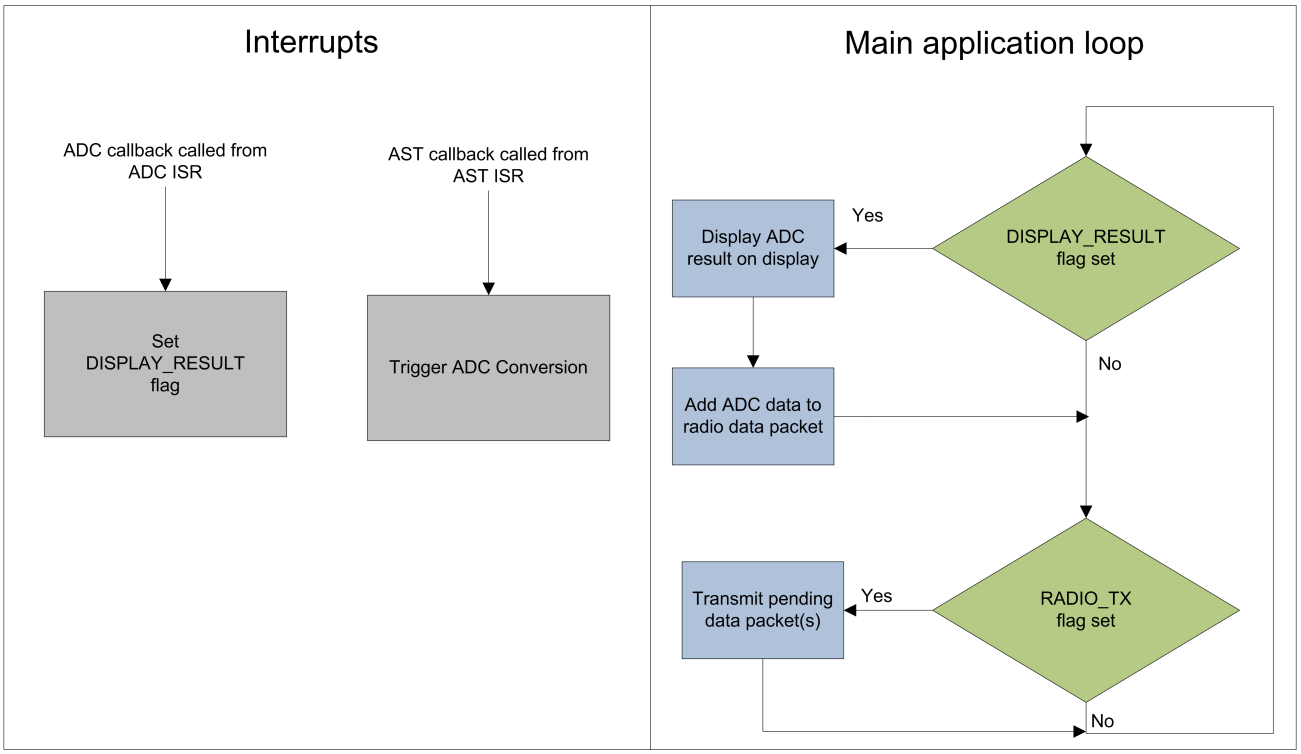
TODO Set the initial hardware configuration on the SAM4L-EK

During this day-long hands-on session you will continue working with the SAM4L-EK. Therefore, once you finish with the thermostat application, it is suggested to set all the jumpers back to the initial configuration.

5.1 Application Flow and Clock Configuration

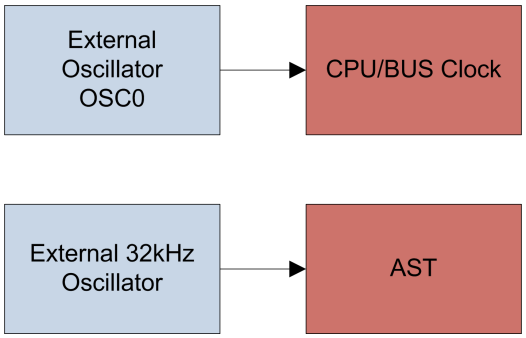
Figure 5-1: Application flow chart shows a flow diagram of the completed application, where you can see how the main application and the interrupts interact. The ADC sample buffer is filled each second in the AST interrupt routine. When the data is ready, the main application is signaled so that the data is displayed on the LCD and transmitted to the head-unit through the wireless network.

Figure 5-1. Application flow chart



In addition, Figure 5-2: Clock configuration shows how the CPU and the AST are clocked in the application that we have built.

Figure 5-2. Clock configuration



5.2 Code Summary

This is a summary of all the code that has been added during this training:

```
#include <asf.h>
#include <string.h>
#include "temp_sensor.h"
#include "data_protocol.h"
#include "avr2025_mac.h"
#include "pal.h"

#define APP_ADC_SAMPLES    1
uint16_t g_adc_sample_data[APP_ADC_SAMPLES];

#define PROTOCOL_ADDRESS 0x32


enum protocol_channels {
    PROTOCOL_LIGHT,
};

volatile bool radio_ready = false;

enum app_state {
    APP_STATE_RADIO_TX,
    APP_STATE_DISPLAY_RESULT,
};

volatile uint16_t app_state_flags = 0;

static bool is_app_state_set(
        enum app_state state)
{
    bool retval;

    cpu_irq_disable();

    if (app_state_flags & (1 << state)) {
        retval = true;
    } else {
        retval = false;
    }

    cpu_irq_enable();
    return retval;
}
```

```
static void set_app_state(
        enum app_state state)
{
    cpu_irq_disable();

    /* Set corresponding flag */
    app_state_flags |= (1 << state);

    cpu_irq_enable();
}

static void clear_app_state(
        enum app_state state)
{
    cpu_irq_disable();

    /* Clear corresponding flag */
    app_state_flags &= ~(1 << state);

    cpu_irq_enable();
}

struct adc_dev_inst g_adc_inst;

static void adcife_read_conv_result(void)
{
    // Check the ADC conversion status
    if ((adc_get_status(&g_adc_inst) & ADCIFE_SR_SEOC) == ADCIFE_SR_SEOC){
        g_adc_sample_data[0] = adc_get_last_conv_value(&g_adc_inst);
        adc_clear_status(&g_adc_inst, ADCIFE_SCR_SEOC);
        set_app_state(APP_STATE_DISPLAY_RESULT);
    }
}


static void adc_setup(void)
{
    struct adc_config adc_cfg = {
        /* System clock division factor is 16 */
        .prescal = ADC_PRESCAL_DIV16,
        /* The APB clock is used */
        .clksel = ADC_CLKSEL_APBCLK,
        /* Max speed is 150K */
        .speed = ADC_SPEED_150K,
        /* ADC Reference voltage is 0.625*VCC */
        .refsel = ADC_REFSEL_1,
        /* Enables the Startup time */
        .start_up = CONFIG_ADC_STARTUP
    };
    struct adc_seq_config adc_seq_cfg = {
        /* Select Vref for shift cycle */
        .zoomrange = ADC_ZOOMRANGE_0,
        /* Pad Ground */
        .muxneg = ADC_MUXNEG_1,
        /* Scaled Vcc, Vcc/10 */
        .muxpos = ADC_MUXPOS_6,
        /* Enables the internal voltage sources */
        .internal = ADC_INTERNAL_2,
        /* Disables the ADC gain error reduction */
        .gcomp = ADC_GCOMP_DIS,
        /* Disables the HWLA mode */
```

```
        .hwla = ADC_HWLA_DIS,
        /* 12-bits resolution */
        .res = ADC_RES_12_BIT,
        /* Enables the single-ended mode */
        .bipolar = ADC_BIPOLAR_SINGLEENDED
    };

    struct adc_ch_config adc_ch_cfg = {
        .seq_cfg = &adc_seq_cfg,
        /* Internal Timer Max Counter */
        .internal_timer_max_count = 60,
        /* Window monitor mode is off */
        .window_mode = 0,
        .low_threshold = 0,
        .high_threshold = 0,
    };

    adc_init(&g_adc_inst, ADCIFE, &adc_cfg);
    adc_enable(&g_adc_inst);
    adc_ch_set_config(&g_adc_inst, &adc_ch_cfg);
    adc_set_callback(&g_adc_inst, ADC_SEQ_SEOC, adcife_read_conv_result,
    ADCIFE_IRQn, 1);
}



static void ast_setup(void)
{
    ast_enable(AST);

    struct ast_config ast_conf;
    ast_conf.mode = AST_COUNTER_MODE;
    ast_conf.osc_type = AST_OSC_32KHZ;
    /* Prescaler that gives an output of 1kHz */
    ast_conf.psel = 4;

    ast_conf.counter = 0;

    /* Enable osc32 oscillator*/
    if (!osc_is_ready(OSC_ID_OSC32)) {
        osc_enable(OSC_ID_OSC32);
        osc_wait_ready(OSC_ID_OSC32);
    }

    /* Initialize the AST */
    if (!ast_set_config(AST, &ast_conf)) {
        ioport_set_pin_level(LED0_GPIO, LED0_ACTIVE_LEVEL);
        while(true) {
        }
    }
}


static void ast_callback(void)
{
    ast_clear_interrupt_flag(AST, AST_INTERRUPT_ALARM);
    ioport_toggle_pin_level(LED0_GPIO);
    adc_start_software_conversion(&g_adc_inst);
}
```

```
static void ast_callback_setup(void)
{
    /* Callback to send data over wireless every second */
    ast_write_alarm0_value(AST, 1024);
    ast_enable_counter_clear_on_alarm(AST, 0);
    ast_write_counter_value(AST, 0);
    ast_set_callback(AST, AST_INTERRUPT_ALARM, ast_callback,
            AST_ALARM_IRQn, 0);
}

wpan_addr_spec_t dst_addr = {
    .AddrMode = WPAN_ADDRMODE_SHORT,
    .PANId = DESTINATION_PAN_ID,
    .Addr.short_address = DESTINATION_SHORT_ADDR,
};

static void send_data(
        uint8_t *data,
        uint8_t size)
{
    static uint8_t msduHandle = 0;

    wpan_mcps_data_req(WPAN_ADDRMODE_SHORT, &dst_addr,
            size, data, msduHandle++, WPAN_TXOPT_ACK);
}

static void alert(void)
{
    cpu_irq_disable();
    ioport_set_pin_level(LED0_GPIO, LED0_ACTIVE_LEVEL);

    while (true) {
        /* Do nothing */
    }
}

int main(void)
{
    irq_initialize_vectors();
    sysclk_init();
    board_init();

    /* Initialize timer */
    ast_setup();
    ast_callback_setup();

    c42364a_init();
    LED_On(LCD_BL);

    //uint8_t const scrolling_str[] = "HELLO WORLD!";
    //c42364a_text_scrolling_start(scrolling_str,
    //      strlen((char const *) scrolling_str));

    adc_setup();

    ioport_set_pin_dir(AT86RFX_RST_PIN, IOPORT_DIR_OUTPUT);
    ioport_set_pin_level(AT86RFX_RST_PIN, IOPORT_PIN_LEVEL_HIGH);
    ioport_set_pin_dir(AT86RFX_SLP_PIN, IOPORT_DIR_OUTPUT);
    ioport_set_pin_level(AT86RFX_SLP_PIN, IOPORT_PIN_LEVEL_HIGH);
```

```
    uint8_t string_buf[8];
    snprintf(string_buf, 8, "No 0x%2X", PROTOCOL_ADDRESS);
    c42364a_write_alphanum_packet(string_buf);

    delay_init();

    sw_timer_init();

    if (wpan_init() != MAC_SUCCESS) {
        alert();
    }

    cpu_irq_enable();
    wpan_mlme_reset_req(true);
    while (!radio_ready) {
        wpan_task();
    }

    protocol_tx_init(send_data, PROTOCOL_ADDRESS);

    while (true) {
        wpan_task();

        if (is_app_state_set(APP_STATE_RADIO_TX)) {
            clear_app_state(APP_STATE_RADIO_TX);
            protocol_send_packet();
        }

        if (is_app_state_set(APP_STATE_DISPLAY_RESULT)) {
            clear_app_state(APP_STATE_DISPLAY_RESULT);
            c42364a_show_numeric_dec(g_adc_sample_data[0]);
            protocol_set_channel_data(PROTOCOL_LIGHT, &g_adc_sample_data[0]);
            set_app_state(APP_STATE_RADIO_TX);
        }
    }
}
```

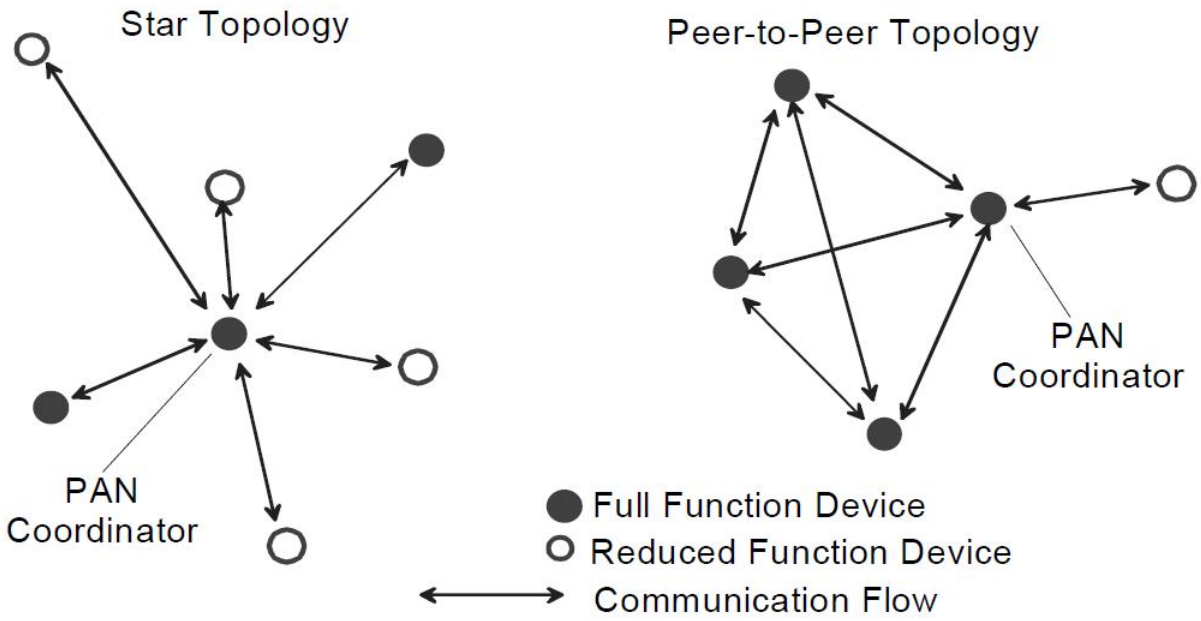
A. Introduction to IEEE802.15.4 Standard

This Appendix gives a brief introduction to IEEE802.15.4 Standard, network topologies, association and disassociation processes, and data transfer models.

A.1 Introduction to IEEE802.15.4 Standard

IEEE 802.15.4 is a standard which specifies the physical layer and media access control for low-rate wireless personal area networks (LR-WPANs). It is maintained by the IEEE 802.15 working group. It is the basis for the ZigBee, RF4CE, and Wireless HART, specifications, each of which further extends the standard by developing the upper layers which are not defined in IEEE 802.15.4. IEEE 802.15.4 defines the physical layer (PHY) and Medium Access Control (MAC) sublayer specifications for low-data-rate wireless connectivity with fixed, portable, and moving devices with no battery or very limited battery consumption requirements typically operating in the personal operating space (POS). A system conforming to this standard consists of several components. The most basic is the Device. A Device may be an RFD or an FFD. Two or more Devices within a POS communicating on the same physical channel constitute a WPAN. Depending on the application requirements, an IEEE 802.15.4 LR-WPAN may operate in either of two topologies: the star topology or the peer-to-peer topology as shown in Figure A-1: Network Topology

Figure A-1. Network Topology



Each independent PAN selects a unique identifier. This PAN identifier allows communication between Devices within a network using short addresses and enables transmissions between Devices across independent networks. All 802.15.4 Devices have a 64-bit (long) IEEE address, which uniquely identifies the Device. In order to extend battery life, shorter addresses are used to shorten the packet sizes and hence the time a Device is actively communicating. All IEEE 802.15.4 based networks use beacons from a Coordinator when joining Devices to the network. The beacon is an IEEE802.15.4 specified frame type which is used by Coordinators to transmit beacon regularly for synchronizing the clock of all the Devices within the same network. It is used by Coordinator to let a specific Device in a network know there is data pending for that Device in the Coordinator and it is also used by Device to locate the Coordinator. In normal operation, an IEEE 802.15.4 based network can operate with or without regular communication beacons and are called beacon-enabled and non-beacon-enabled mode respectively. In beacon-enabled mode, the Coordinator sends out a periodic train of beacon signals containing information that allows network nodes to synchronize their communications. A beacon also contains information on the data pending for the different nodes of the network. In non-beacon-enabled mode, beacons are not transmitted on a regular basis by the Coordinator (but can still be requested for the purpose of associating a Device with the Coordinator). The Device communicates with the Coordinator only when it needs. To determine whether there

is data pending for a node, the node must poll the Coordinator (in a beacon-enabled network, the availability of pending data is indicated in the beacons).

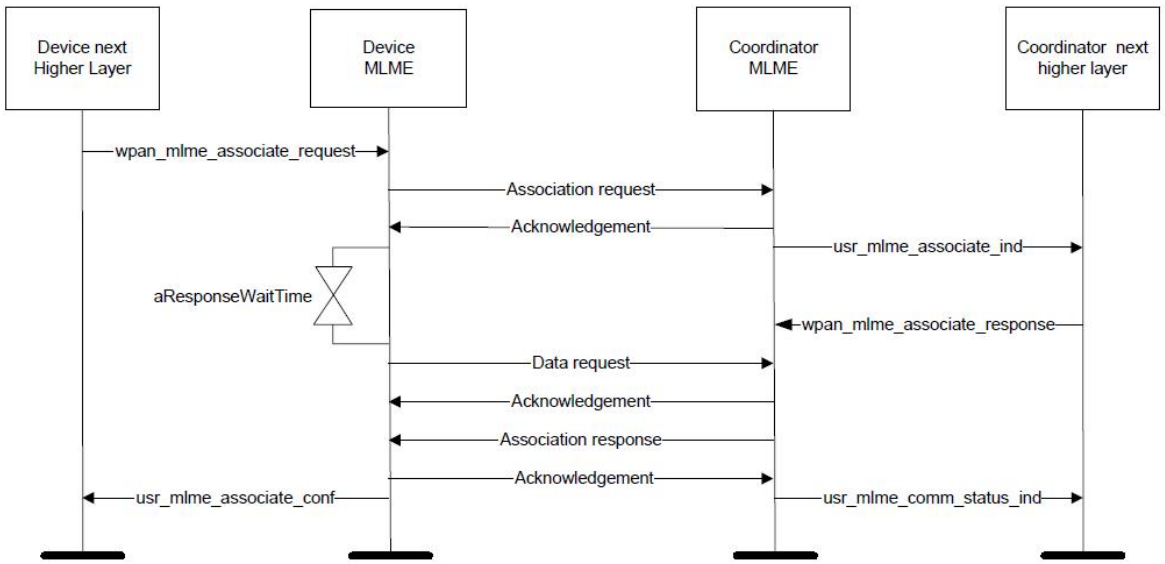
A.2 Association and Disassociation Processes

The section below explains the procedure for association and dissociation of a Coordinator and Device Node.

A.2.1 Association Process

On startup, the Coordinator Node and Device Node will reset the MAC sublayers by calling MLME-RESET.request primitive (MLME - MAC sublayer management entity). When the higher layer (the higher layer can be an application layer in a MAC based projects or networking layer in RF4CE / ZigBee based projects) receives a reset confirmation, the default PIB attributes (PAN Information Base) values are loaded. The Device Node, on confirmation, will do a channel scan and find its PAN Coordinator, ie the Device will send a beacon request command over the air and scan for the reception of beacon frame from the Coordinator. On selection of the PAN Coordinator with which to associate, the higher layers will send association request to Coordinator node request using MLME-ASSOCIATE.request primitive. The Coordinator node on receiving the association request will inform the higher layer and this layer decide whether the Device should be allowed to join the network. This decision is taken within a time frame macResponseWaitTime. If sufficient resources are available the higher layer in the Coordinator Node assign a short address to the Device and MAC will generate the association response command and send to Device Node using indirect data transmission. In Device Node, on receipt of association response, the MAC layer inform the higher layer about the association confirmation as shown in Figure A-2: Association Process.

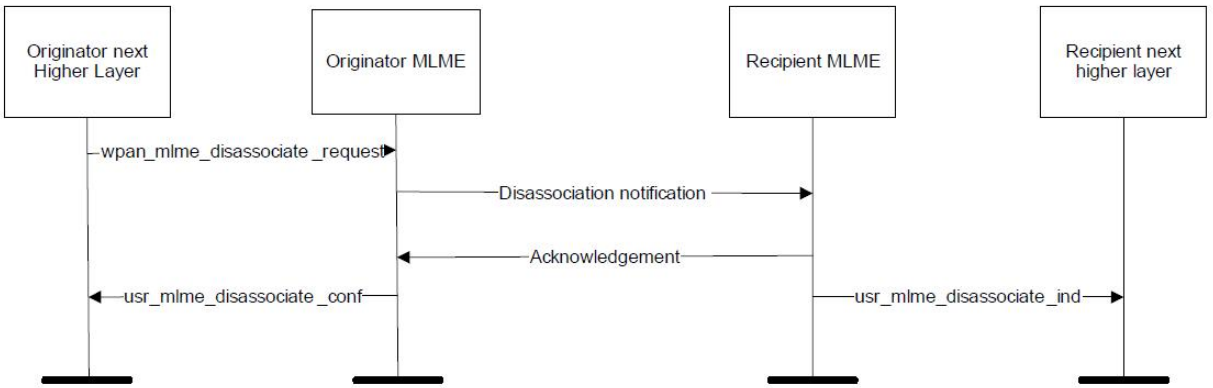
Figure A-2. Association Process



A.2.2 Disassociation Process

The Disassociation procedure is initiated by the higher layer by issuing the MLME-DISASSOCIATE.request primitive. When the higher layer in the Coordinator wants one of its associated Devices to leave the PAN, the higher layer of the Coordinator will send the disassociation notification command to the MAC sublayer in the primitive MLME-DISASSOCIATE.request. The MLME of the Coordinator will send the disassociation notification command to the Device using indirect transmission. Once the command is sent, the higher layer is informed. On receipt of the disassociation request the Device Node will also inform the higher layer about the disassociation. If the associated Device wants to leave PAN network, the MAC sublayer of the Device will send a disassociation notification command using the direct transmission as shown in Figure A-3: Disassociation Process.

Figure A-3. Disassociation Process



A.3 Data Transfer Model

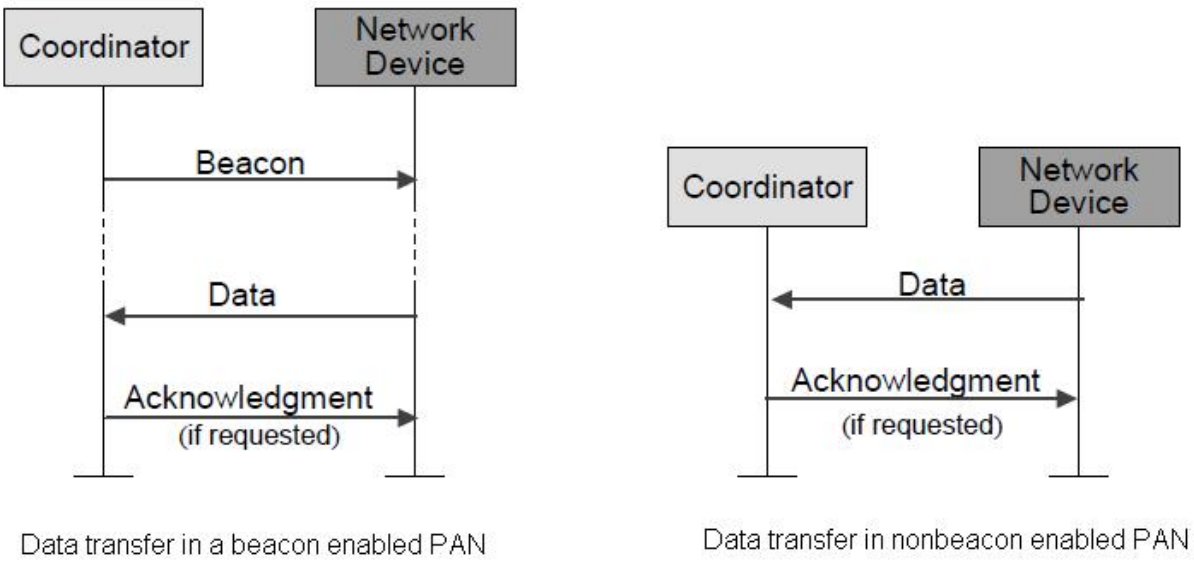
The IEEE802.15.4 defines three types of data transactions: The IEEE802.15.4 defines three types of data transactions

- Data transfer to a Coordinator
- Data transfer from a Coordinator
- Peer to Peer data transfer

A.3.1 Data transfer to a Coordinator

When a Device wishes to transfer data to a Coordinator in a beacon-enabled PAN, it first listens for the network beacon and synchronize to it. At the allocated time slot, the Device transmits its data frame to the Coordinator. The Coordinator may acknowledge the successful reception of the data by transmitting an optional acknowledgment frame. When a Device wishes to transfer data in a non-beacon-enabled PAN, it simply transmits its data frame to the Coordinator. The Coordinator acknowledges the successful reception of the data by transmitting an optional acknowledgment frame. Figure A-4: Data transfer to a Coordinator shows the data transfer form a Device Node to Coordinator Node.

Figure A-4. Data transfer to a Coordinator



A.3.2 Data transfer from a Coordinator

When the Coordinator wishes to transfer data to a Device in a beacon-enabled PAN, it indicates in the network beacon that the data message is pending. The Device periodically listens to the network beacon and, if a message is pending, transmits a MAC command requesting the data. The Coordinator acknowledges the successful reception of the data request by transmitting an acknowledgment frame. The pending data frame is then sent, if possible, immediately after the acknowledgment. The Device may acknowledge the successful reception of the data by transmitting an optional acknowledgment frame. When a Coordinator wishes to transfer data to a Device in a non-beacon-enabled PAN, it stores the data for the appropriate Device to make contact and request the data. When the Device makes contact by transmitting a MAC command requesting the data to its Coordinator, the Coordinator acknowledges the successful reception of the data request by transmitting an acknowledgment frame. If a data frame is pending, the Coordinator transmits the data frame to the Device. If a data frame is not pending, the Coordinator indicates this fact either in the acknowledgment frame following the data request or in a data frame with a zero length payload. Figure A-5: Data transfer from a Coordinator shows the data transfer from Coordinator Node to the Device Node.

Figure A-5. Data transfer from a Coordinator

A.3.3 Peer-to-Peer Data Transfer

In a peer-to-peer PAN, every Device may communicate with every other Device in its radio sphere of influence. In order to do this effectively, the Devices wishing to communicate will need to either receive constantly or synchronize with each other.

B. ASF User Guide

The following sections describe drivers, Quick Start guides, version information and license:

- Getting Started with ASF
  - License

B.1 Getting Started with ASF

The intention of ASF is to provide a rich set of proven drivers and code modules developed by Atmel experts to reduce customer design-time. It simplifies the usage of microcontrollers, providing an abstraction to the hardware and high-value middlewares.

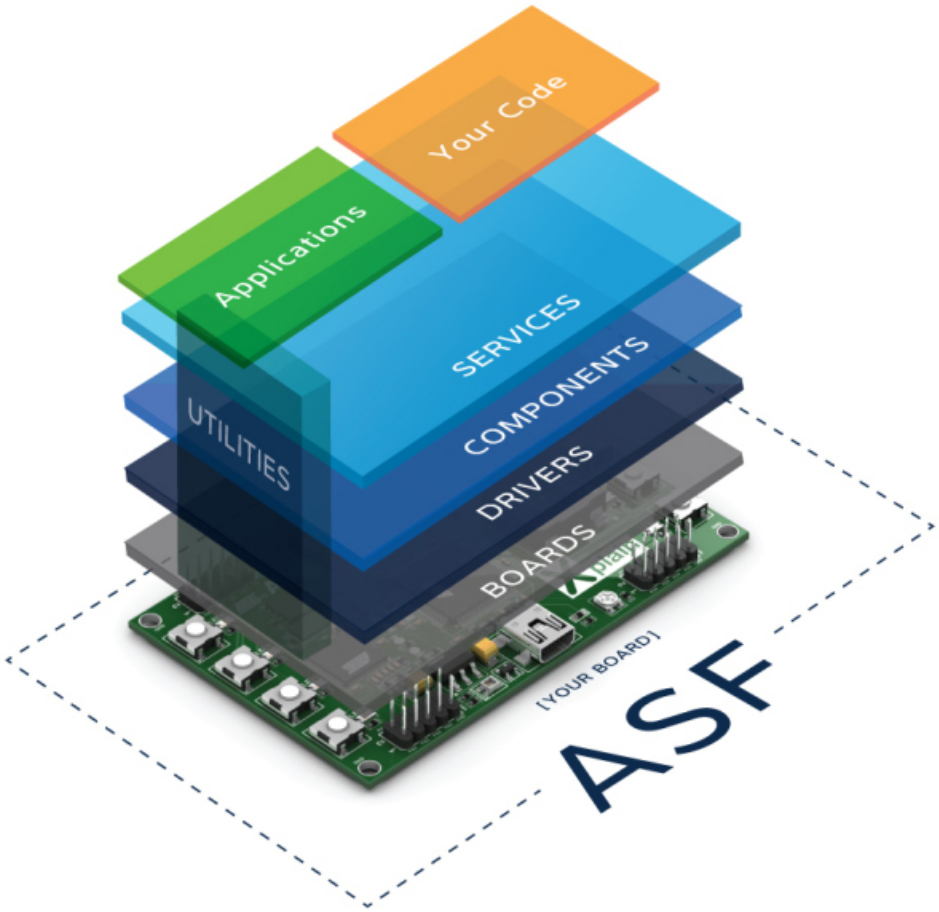
ASF is a free and open-source code library designed to be used for evaluation, prototyping, design and production phases.

- ASF is code-size-optimized:
  - Multiple ANSI-C compilers supported.
  - Architecture-optimized by Atmel experts.
- ASF is performance-optimized:
  - DMA for communication.
  - Interrupt-driven drivers.
  - Chip-specific features in stacks.
- ASF is low-power-optimized:
  - Clock masking API, Sleep management API.
  - Support for hardware SleepWalking™ Event controller.
  - Dynamic frequency and voltage scaling.

B.1.1 Architecture

ASF consists of source code modules and applications demonstrating the use of these.

Figure B-1. ASF Architecture



- Drivers is composed of a driver.c and driver.h file that provides low level register interface functions to access a peripheral or device specific feature. The services and components will interface the drivers.
- Services is a module type which provides more application oriented software such as a USB classes, FAT file system, architecture optimized DSP library, graphical library, etc.
- Components is a module type which provides software drivers to access external hardware components such as memory (e.g. Atmel DataFlash®, SDRAM, SRAM, and NAND flash), displays, sensors, wireless, etc.
- Boards contains mapping of all digital and analog peripheral to each I/O pin of Atmel's development kits.

B.1.2 About

Documentation is generated using Doxygen, and all the Doxygen tags/comments are therefore in the source code, also to make it possible for you to reuse this documentation in your own project if desired.

ASF is currently oriented towards Atmel Studio[1] , since it contains so much code that it can be hard to navigate in it manually. Atmel Studio is offering a helping hand through the use of wizards, filters and documentation.

For those of you who do not want to use Atmel Studio, you can download and use ASF as a stand-alone package, see the download page. Note that ASF also supports IAR and have IAR project files for all projects.

[1] www.atmel.com/atmelstudio

B.2 License