
How to implement SleepWalking on an ARM Cortex-M4 MCU Application: Step-by-step Project Building Guide

Atmel ARM Cortex-M4 SAM4L Product Family

Introduction

The aim of this document is to provide a step by step guide on how to implement SleepWalking on an Atmel® ARM® Cortex®-M4 MCU, from the SAM4L product family.

By using Atmel Studio 6 and the Atmel Software Framework (ASF), the user will have a complete application example allowing interactions with touch and light sensors and the LCD Display of the Atmel SAM4L-EK board.

In order to save a maximum of energy, the SleepWalking allows the CPU to sleep peacefully until a relevant event occurs. To be able to perform SleepWalking, the Atmel SAM4L is a picoPower® branded ARM MCU, which embeds power scaling and power saving capabilities, coupled with a high flexibility in terms of clock management. These features are mandatory to implement SleepWalking in the most advantageous way.

As the SleepWalking is just an optimization in using low power modes; this can be used over any kind of application. In this example a real time watch application is proposed.

Finally, thanks to this application, full examples on how to use the different embedded peripherals involved into the SleepWalking are covered, such as the Peripheral Event Controller (PEVC), the hardware QTouch® (CATB), the ADC window mode (ADCIFE), the External Interrupt Controller (EIC) and the Asynchronous Timer (AST).

Prerequisites

- Hardware prerequisites
 - Atmel SAM4L-EK Evaluation kit
 - Micro-USB cable – 1 No.
- Software prerequisites
 - Atmel Studio 6.1 (build 2730) or higher
 - Atmel Software Framework 1.13.1 or higher

Table of Contents

Introduction	1
Prerequisites.....	1
1 Introduction to SleepWalking	4
1.1 On what does SleepWalking consist in?	4
1.2 SAM4L Features to Perform SleepWalking.....	4
1.2.1 Low Power Techniques Overview	4
1.2.2 Peripheral Clock Management Overview	6
1.2.3 Peripheral Event System Controller (PEVC) Overview	6
2 SleepWalking Application Description	9
2.1 Atmel SAM4L-EK Evaluation Kit Overview	9
2.1.1 Powering the Board.....	9
2.1.2 The Board Monitor.....	10
2.2 Application Overview.....	10
2.2.1 Application Sequential Flowchart	12
2.2.2 Function Descriptions.....	12
2.2.2.1 ACTIVE_MODE.....	13
2.2.2.2 INIT_SLEEPWALKING.....	13
2.2.2.3 SLEEPWALKING	14
2.2.2.4 INIT_ACTIVE_MODE	14
2.3 Software Package Description	14
2.4 Main Steps Summary.....	14
3 Atmel Studio 6 Project Creation	16
3.1 Atmel Studio Introduction	16
3.1.1 IDE Introduction	16
3.1.2 Atmel Software Framework (ASF) Introduction	17
3.2 Create a Project under Atmel Studio using a Specific Board Template	17
3.3 Add Existing File to the Project	20
3.4 Atmel Software Framework Module Importation	22
3.5 Debugger Settings	27
4 System Initialization.....	29
4.1 Starting Point.....	29
4.1.1 Include Example Library File	29
4.2 Application Init Function Implementation.....	29
4.3 Application's Clock Setting Configuration.....	30
5 SAM4L Peripherals Configuration Method	35
5.1 Application Peripheral Configuration Overview	35
5.2 Methodology.....	35
6 Analog to Digital Converter (ADCIFE) Configuration.....	37
6.1 Check any Hardware Configuration Related to the Peripheral.....	37
6.2 Use the Online API Documentation which provides Simple Example on ADCIF	37
6.3 Code Implementation	41

6.3.1	Configure the ADCIFE using the Quick Start Guide Workflow	41
6.3.1.1	Define the ADCIFE Interrupt Handler in the Application	41
6.3.1.2	Configure ADC Module.....	42
7	Capacitive Touch (CATB) Configuration	45
7.1	Autonomous Touch Overview	45
7.2	Code Implementation	45
8	Asynchronous Timer (AST) Configuration	48
8.1	Code Implementation	48
9	Peripheral Event System Controller (PEVC) Configuration.....	51
9.1	Code Implementation	51
10	External Interrupt Controller (EIC) Configuration	54
10.1	Code Implementation	54
11	Test the App Init Function Implementation	57
12	State Machine Implementation	59
12.1	Code Implementation	59
12.1.1	Implement the App Function	59
12.1.2	ACTIVE_MODE Implementation	60
12.1.3	Prepare to SleepWalking Implementation	61
12.1.3.1	Code Implementation	61
12.1.4	SLEEPWALKING State Implementation	62
12.1.4.1	Code Implementation	62
12.1.5	INIT_ACTIVE_MODE State Implementation	63
12.1.5.1	Code Implementation	64
12.1.6	Update the Main Function to Call the App Function	65
13	Conclusion.....	66
14	Suggested Reading	67
14.1	Device Datasheet.....	67
14.2	Evaluation Kit User Guide	67
14.3	ARM Documentation on Cortex-M4 Core.....	67
15	Revision History	68

1 Introduction to SleepWalking

1.1 On what does SleepWalking consist in?

As part of the Atmel picoPower technology SleepWalking adds intelligence to the SAM4L peripherals. This allows a peripheral to determine if incoming data requires use of the CPU or not. **We call this SleepWalking** because it allows the CPU to sleep peacefully until a relevant event occurs.

In the traditional way of addressing this, the internal timer wakes up the microcontroller periodically to check whether certain conditions that require its attention are present or not. The CPU and RAM traditionally consume the majority of the power in active mode, and so waking up the CPU to check for these conditions will consume a lot of power in the long run. In some cases where the reaction time is too short, it might not even be possible for the CPU to go back into sleep mode at all.

The Atmel SAM4L microcontroller solves this problem with its SleepWalking peripherals. SleepWalking allows the microcontroller to be put into deep sleep and wake up only upon a pre-qualified event. The CPU no longer needs to check whether or not a specific condition is present, such as an address match condition on the TWI (I²C) interface, or a sensor connected to an ADC that has exceeded a specific threshold.

With SleepWalking, this is done entirely by the peripherals themselves. The CPU and RAM will not wake up until the condition true.



INFO

A great video which perfectly described the SleepWalking is available and could be a good introduction to this application note. This video is available here: [Atmel - picoPower Labs - SleepWalking](#). This video is based on Atmel AVR[®] UC3 microcontrollers but it can be extrapolated to ARM SAM4L microcontrollers.

1.2 SAM4L Features to Perform SleepWalking

SleepWalking allows reducing the total system power consumption in your application.

To perform SleepWalking, the SAM4L has to embed flexible capabilities regarding its peripheral clock management and must be able to get a higher modularity in power consumption versus performance ratio. This is done with its embedded features which the user needs to be familiar with. These features are described below:

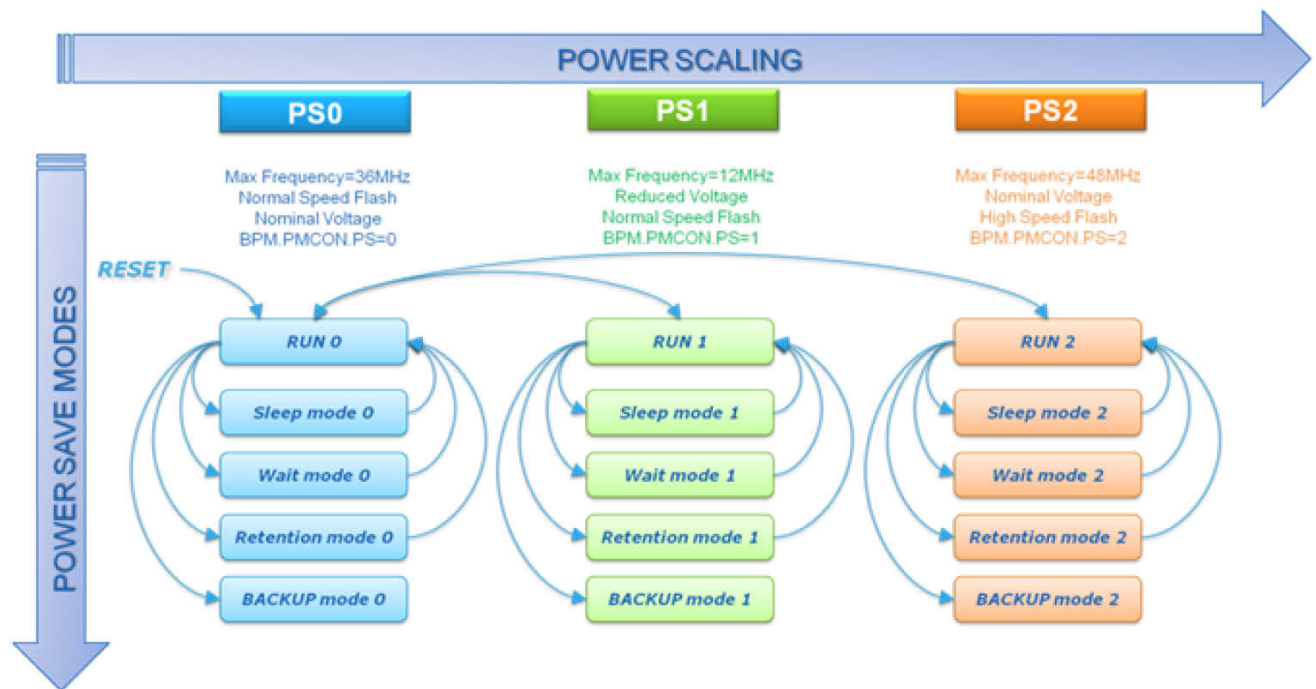
- Low power techniques: Power Saving and Power Scaling
- Peripheral Clock Management flexibility
- Peripheral Event System

1.2.1 Low Power Techniques Overview

The low power techniques are illustrated in the [Figure 1-1](#) and are composed of:

- **Power Save** modes intended to reduce the logic activity and to adapt the power configuration. See “*Power Save Modes*” chapter from the product datasheet. [Table 1-1](#) is taken from the *SAM4L product family datasheet*, and gives a brief description on the power saving mode.
- **Power Scaling** technique consists of adjusting the internal regulator output voltage (voltage scaling) to reduce the power consumption. According to the requirements in terms of performance, operating modes, and current consumption, the user can select the Power Scaling configuration that fits the best with its application. See “*Power Scaling*” chapter from the product datasheet. [Figure 2-1](#) shows what the main Power Scaling modes available are.

Figure 1-1. Low Power Techniques Scheme for SAM4L



Regarding SleepWalking, PS1 is the best Power scaling mode in term of performance/consumption ratio thanks to the internal regulator low power mode.

The Wait mode is the most interesting mode to perform SleepWalking in terms of consumption. All clock sources are stopped; the core and all the peripherals are stopped except the modules running with the 32kHz clock if enabled. This is the lowest power mode where SleepWalking is supported.

Table 1-1. Power Save Mode Configuration Summary

Mode	Mode Entry	Wake up sources	Core domain	Backup domain
SLEEP	WFI SCR.SLEEPDEEP bit = 0 BPM.PMCON.BKUP bit = 0	Any interrupt	CPU clock OFF Other clocks OFF depending on the BPM.PMCON.SLEEP field see "SLEEP mode" on page 52	Clocks OFF depending on the BPM.PMCON.SLEEP field see "SLEEP mode" on page 52
WAIT	WFI SCR.SLEEPDEEP bit = 1 BPM.PMCON.RET bit = 0 BPM.PMCON.BKUP bit = 0	PM WAKE interrupt	All clocks are OFF Core domain is retained	All clocks are OFF except RC32K or OSC32K if running
RETENTION	WFI SCR.SLEEPDEEP bit = 1 BPM.PMCON.RET bit = 1 BPM.PMCON.BKUP bit = 0	PM WAKE interrupt	All clocks are OFF Core domain is retained	All clocks are OFF except RC32K or OSC32K if running
BACKUP	WFI + SCR.SLEEPDEEP bit = 1 + BPM.PMCON.BKUP bit = 1	EIC interrupt BOD33, BOD18 interrupt and reset AST alarm, periodic, overflow WDT interrupt and reset external reset on RESET_N pin	OFF (not powered)	All clocks are OFF except RC32K or OSC32K if running

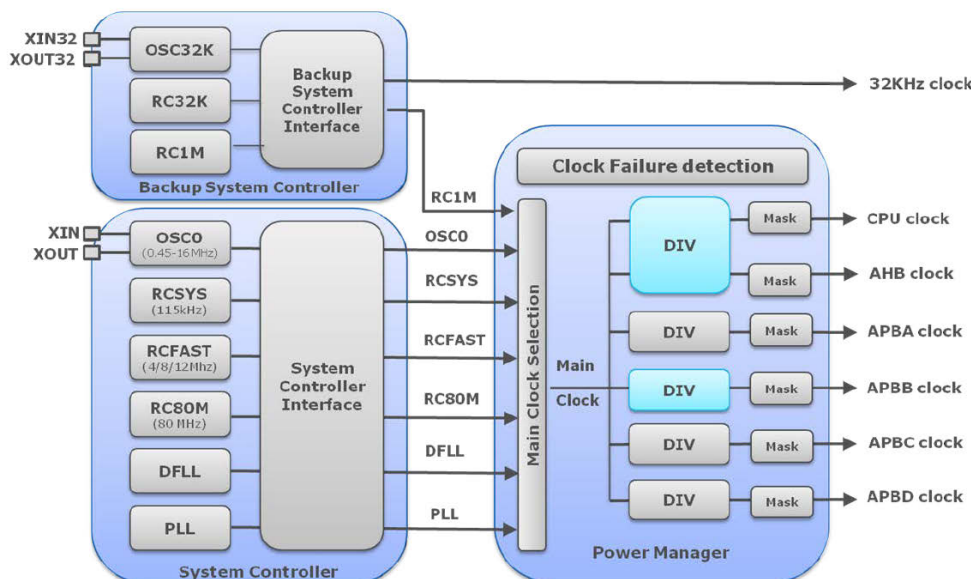
Once the power scaling and the power saving modes are identified to run SleepWalking, the next step is to enable/disable the used/unused peripheral thanks to the SAM4L high flexibility in term of clock management.

1.2.2 Peripheral Clock Management Overview

The SAM4L clock sources depend on two controllers that manage main clock sources and 32kHz clock generation: (see Figure 1-2).

- The Backup system controller which manages and outputs the 32kHz sources and the RC 1MHz
- The system controller which manages all the other main clock sources such as RC FAST, RCSYS, OSC0, PLL and DFLL (new feature on SAM4L product family) and outputs them to the Power manager

Figure 1-2. SAM4L Clock Management Scheme



Regarding SleepWalking activity, the user has to identify what is the main clock source he wants to use in run mode and in Wait mode (during the SleepWalking), according to his application specifications.

Concerning the wait mode, a good choice would be to use the RC FAST oscillator to have the benefit of the SAM4L fast Wake up feature when the core is woken up by the peripheral interrupt.

Once the main clock selection is done in the power manager, each system clock can be enabled through its own mask register and its frequency can also be divided by its own divider.

Regarding the SleepWalking activity, the user has to identify what are the peripheral clocks he wants to use during the SleepWalking according to his application specifications.

For instance, if the AST is used during SleepWalking, its peripheral clock and its source clock must be configured before entering in SleepWalking.

After the clock management, the next step is to maintain some SAM4L peripheral to stay awaked by keeping their clock enabled during the SleepWalking.

The aim is to make measurement or detection while the core is sleeping. Then once an event (e.g.: ADCIFE threshold reached) or a detection (e.g.: In touch CATB sensing detected) occurs, the peripheral wakes up the core with an interrupt related to the event. This peripheral interconnection is done by using the Peripheral Event System Controller.

1.2.3 Peripheral Event System Controller (PEVC) Overview

The Peripheral Event System allows the SAM4L to:

- Manage directly peripheral to peripheral communication

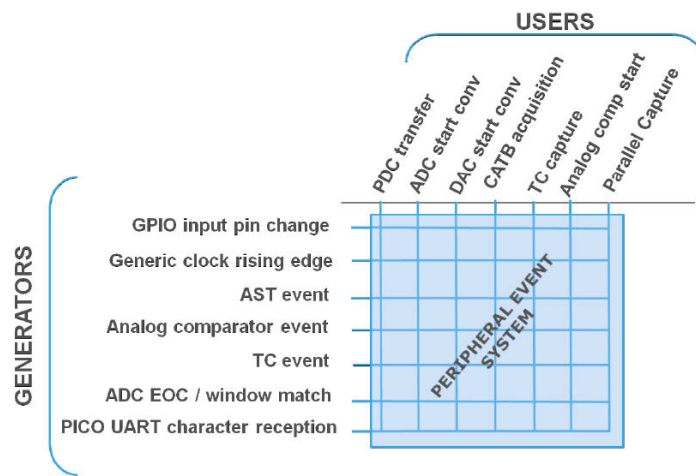
- Receive, react to, and send peripheral events without CPU intervention
- Have cycle deterministic event communication
- Enable SleepWalking and asynchronous interrupts for peripheral operation in Power Save Modes

Several peripheral modules can be configured to emit or respond to signals known as peripheral events. The exact condition to trigger a peripheral event, or the action taken upon receiving a peripheral event, is specific to each module.

Peripherals that respond to events are called **users** and peripherals that emit events are called **generators**.

A module may be both a generator and user. The peripheral event generators and users are interconnected by a network known as the Peripheral Event System as described in [Figure 1-3](#).

Figure 1-3. Peripheral Event System Array Representation



The Peripheral Event Controller (PEVC) controls the interconnection parameters, such as generator-to-user multiplexing and peripheral event enable/disable.

PEVC routes incoming events to users by means of one channel per user. Channels operate in parallel, allowing multiple users to listen to the same generator.

The Channel Multiplexer Register (CHMXi) is written to allocate a generator to a given channel. The channel setting is then enabled by setting a one to the appropriate bit in the Channel Enable Register (CHER). It is disabled by writing a one to the appropriate bit in the Channel Disable Register (CHDR).

To safely configure a channel, user software must follow this process:

- disable the channel by using the CHDR
- Allocate generator by configuring the CHMXi register
- enable the channel by setting the CHER

The PEVC supports asynchronous events in addition to standard synchronous events. Asynchronous events can even be processed in various sleep modes with no running system clock. This is accomplished by taking advantage of some Advanced Power Manager features such as Sleep Walking and Asynchronous Wakeup.

SleepWalking is a particular mode which allows the PEVC to handle asynchronous events in various sleep modes by requesting a module local clock for the duration of the Event processing. Once the event processing is done, the requested clock is disserted and the module goes back to sleep. As a consequence there are some peripherals which are not able to support SleepWalking. [Table 31-7](#) in the [SAM4L product family datasheet](#) specifies what the generators able to perform SleepWalking are, and the [Table 31-8](#) specifies which the users are.

Therefore, before configuring the PEVC, the user must enable peripheral events at generator level and at the User Interface level. Next, the Generator will generate peripheral events periodically, and the Peripheral Event

System will route the peripheral events to the ADC Interface, which will perform ADC conversions or CATB sensing (e.g.) at the selected intervals.



If you want to learn more about the Peripheral Event System, refer to the SAM4L product family datasheet available directly from this web address:

http://www.atmel.com/images/atmel-42023-arm-microcontroller-atsam4l-low-power-lcd_datasheet.pdf.

2 SleepWalking Application Description

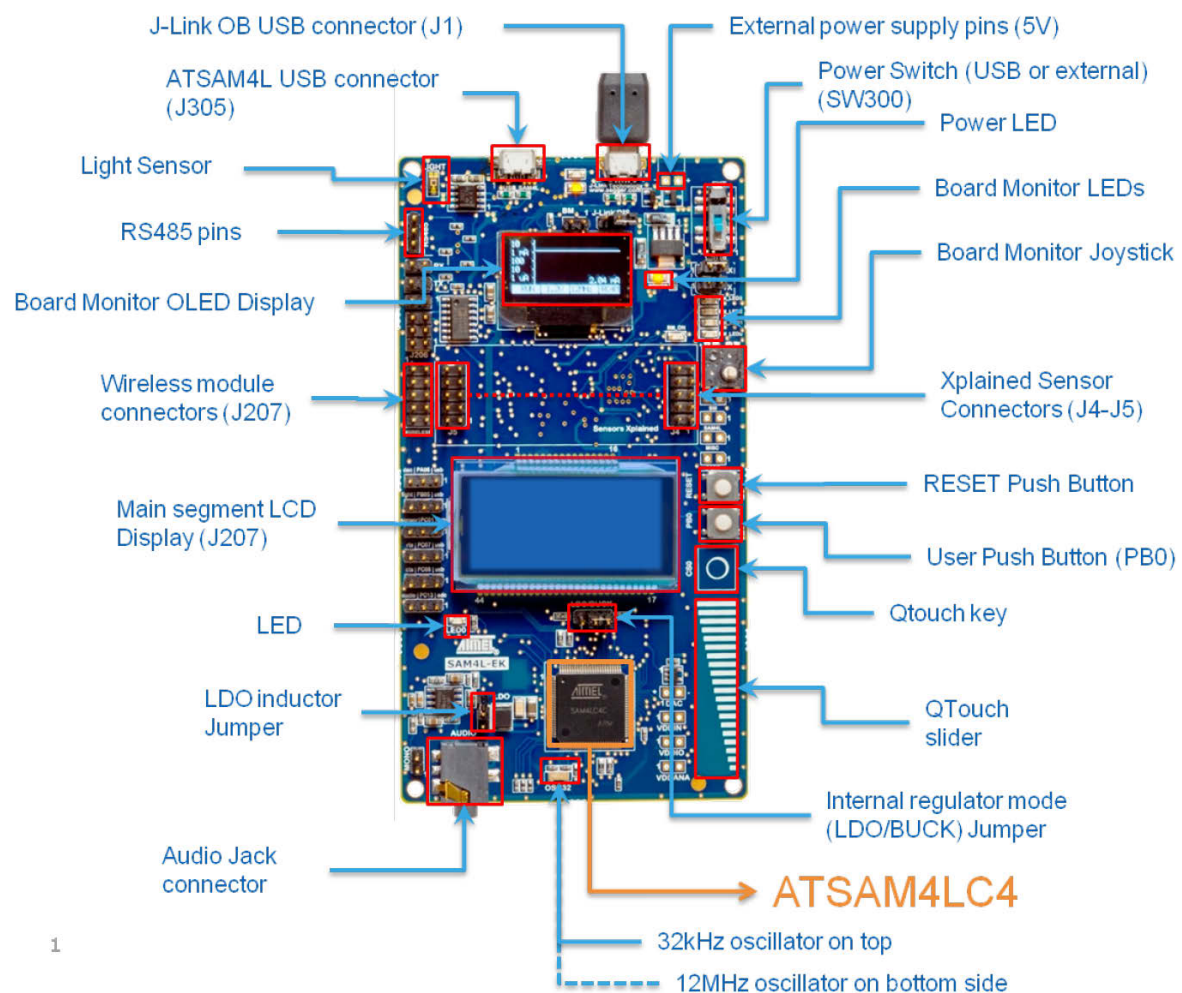
This application note has been built from the Atmel Training documentation, where the hands-on is based on the Atmel SAM4L-EK.

As the SleepWalking is just the definition of a particular ability of the SAM4L during low power modes, we can use it with any application where the current consumption is considered as a key parameter. Regarding this application note, even if a detailed description is given, the main application could be considered as a “Black Box” to stay a maximum focused on the SleepWalking purpose.

This Application note is therefore built to make the reader able to skip the part of the “Black box” application implementation.

2.1 Atmel SAM4L-EK Evaluation Kit Overview

Figure 2-1. ATSAM4L-EK Board Overview



2.1.1 Powering the Board

The ATSAM4L-EK offers three interfaces to power up the board:

- USB embedded debugger SEGGER J-Link OB (J1)
- USB ATSAM4LC4C (J305)
- External 5V (DC) source connected to the J303 2-pin header

To select the USB or the external power supply, a mechanical switch is used (SW300).

The default kit configuration is using ATSAM4LC4C BUCK regulator configuration, where ATSAM4LC4C is powered at 3.3V (VDDIN, VDDIO, VDDANA).

The following board configuration has to be checked:

LDO INDUCTOR JUMPER	OPENED
LDO/BUCK JUMPER	BUCK

The power is supplied through the USB embedded debugger SEGGER J-LINK OB connector (J1).

2.1.2 The Board Monitor

The Board monitor is an on board tool which makes possible the chip current consumption real time monitoring. It is implemented by an on board ATSAM3N4A MCU. The SAM4LC4C sends commands to update the board monitor status through the UART, to the ATSAM3N4A.

The Board Monitor features:

- 1x OLED display (128x64)
- 5x LEDs
- 1x joystick
- 1x USART connected to the ATSAM4LC4C MCU
- 1x TWI connected to the ATSAM4LC4C MCU

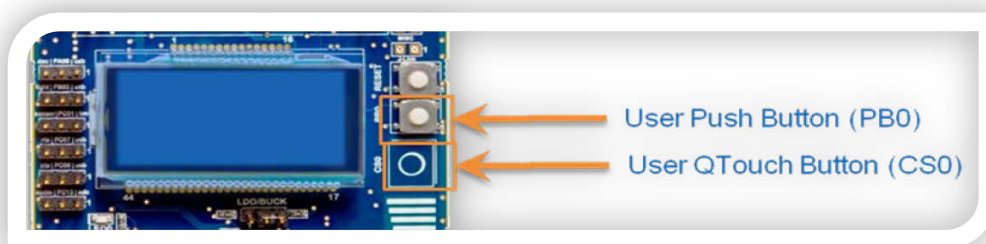
The board monitor will be very useful to see when the CPU is in Low Power mode.

2.2 Application Overview

Any application can be coupled with SleepWalking. In this example a real time watch application is implemented. After the main initialization, the main application is based on a state machine sequence which allows the user to jump from a state to another through interrupt which are directly triggered by acting on:

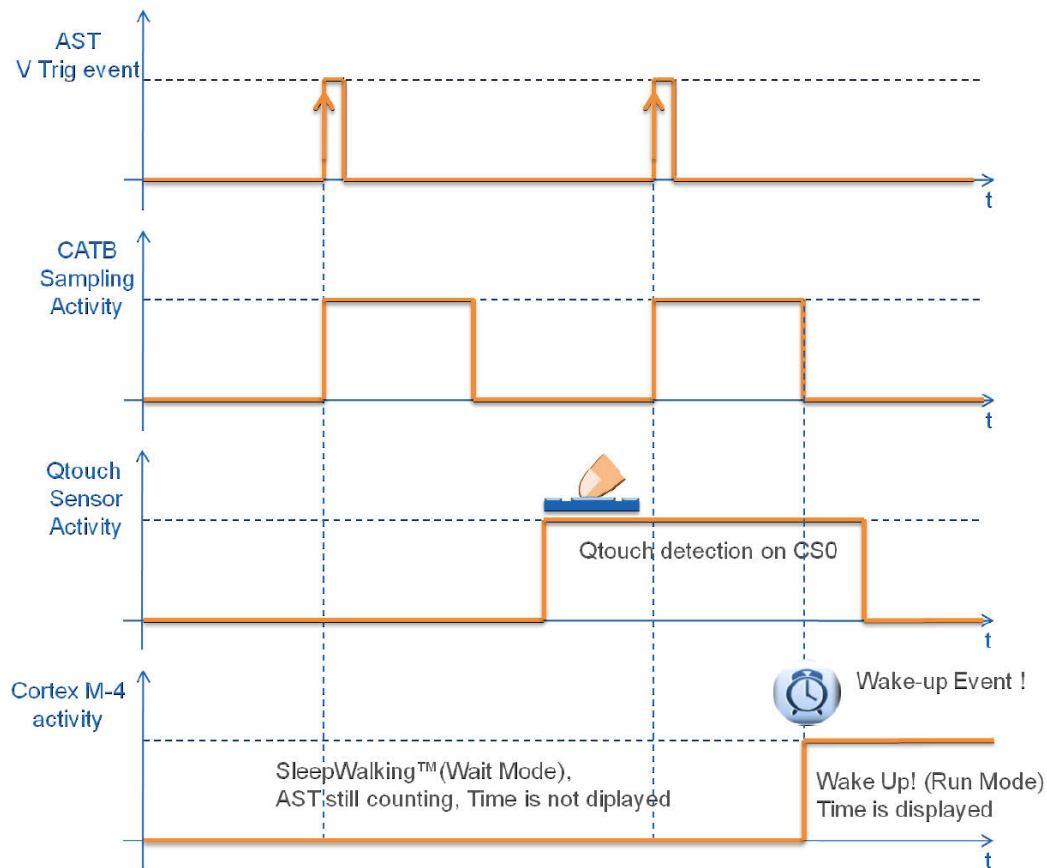
- The QTouch key sensor (CS0) used to wake up the SAM4L device from the WAIT mode using SleepWalking
- The Push Button (PB0) used to go back in WAIT mode with SleepWalking enabled

Figure 2-2. PB0 and CS0 Board Implementation



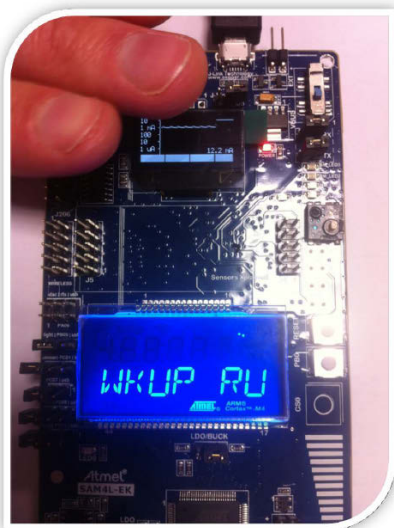
According to this, the timing diagrams are the following:

Figure 2-3. Timing Diagrams on the QTouch Activity



An additional feature is implemented regarding the initial training. As described in the [Figure 2-4](#), the user will be able to exit from power saving mode thanks to the embedded light sensor which is directly connected to an ADC input.

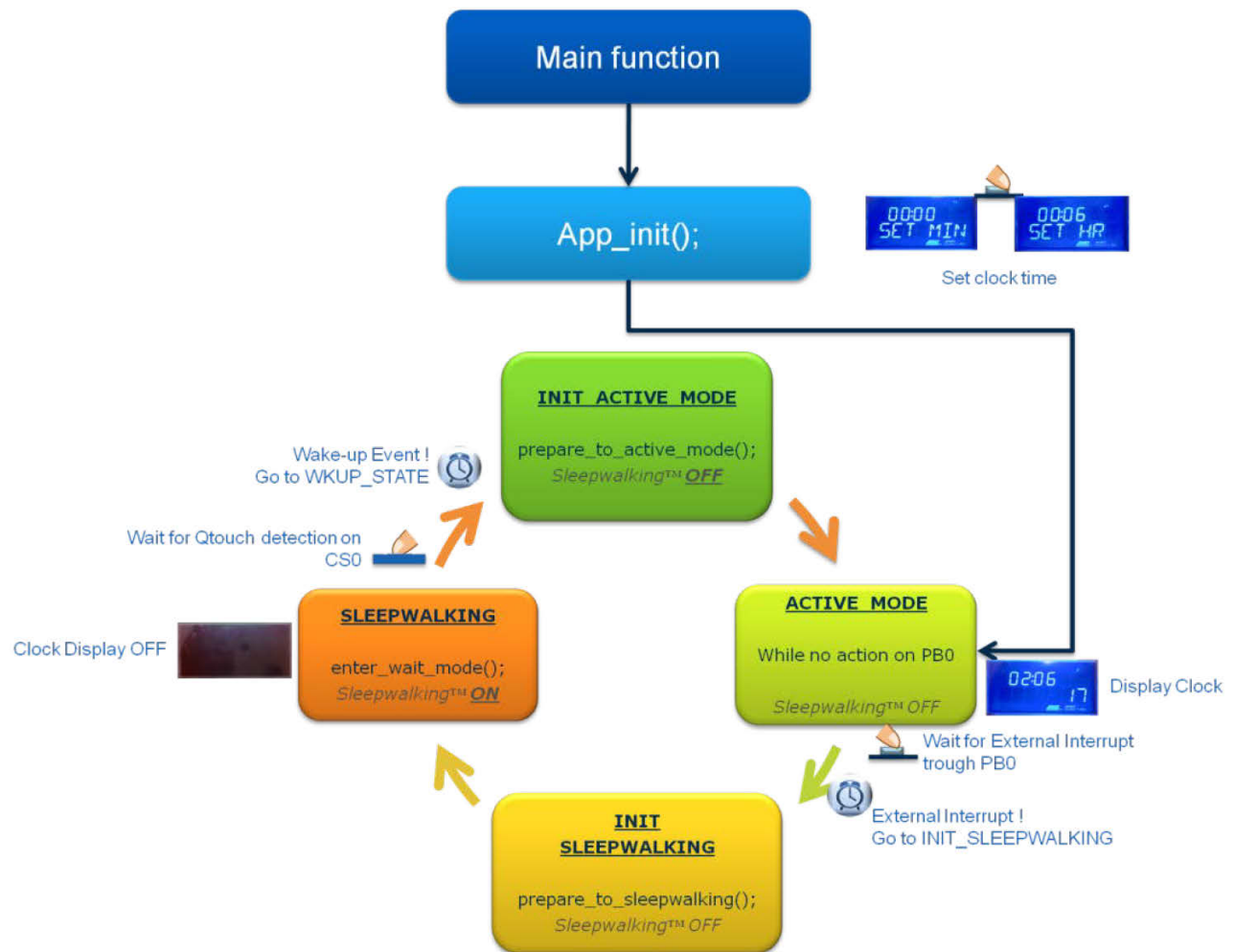
Figure 2-4. Wake the System up thanks to the Light Detection



2.2.1 Application Sequential Flowchart

Figure 2-5 sums up the SAM4L SleepWalking application flowchart.

Figure 2-5. SleepWalking Application Sequential Flowchart

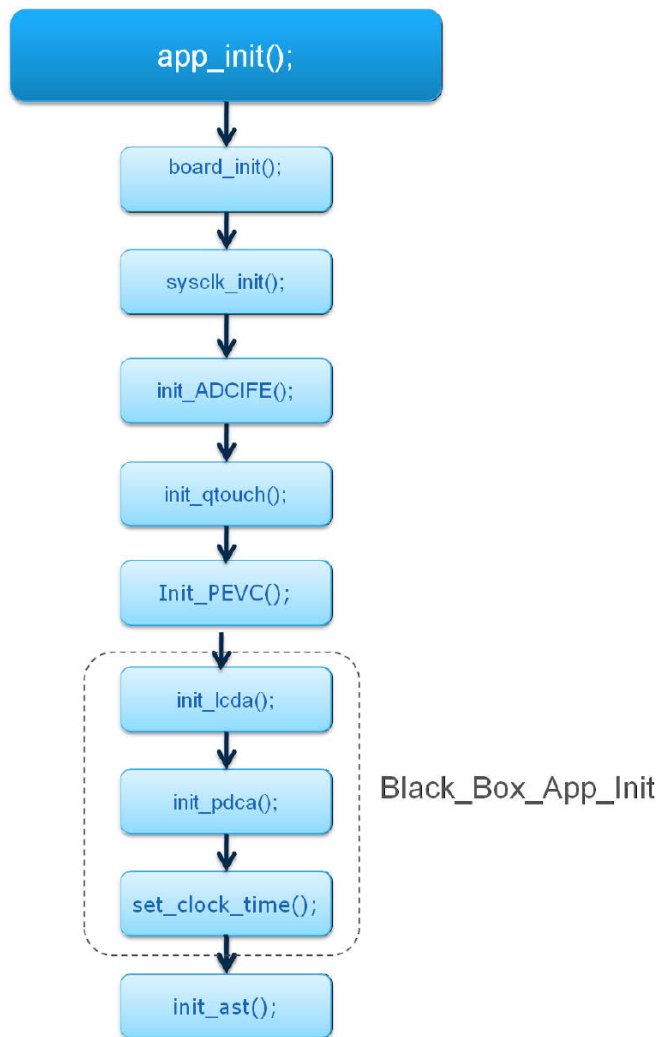


2.2.2 Function Descriptions

The application initialization (*app_init()*) function is called after a power up and executes the following functions:

- `sysclk_init()`: The main clock is the PLL using the main oscillator running @48MHz
- `board_init()`: Initializes the SAM4L-EK board
- `init_qtouch()`: Initializes the Capacitive Touch (CATB) Module and the QTOUCH Library
- `init_lcd()`: Initializes the Segment LCD Controller (LCDCA) to display the clock time
- `init_pdca()`: Initializes the Peripheral DMA (PDCA)
- `set_clock_time()`: Set Clock Time
- `CATB_PEVc_init()`: Initializes the CATB module as a SleepWalking User peripheral with the AST as generator
- `PDCA_PEVc_init()`: Initializes the PDCA module as another SleepWalking User peripheral with the AST as generator
- `init_ast()`: Initializes the AST in calendar mode

Figure 2-6. app_init Flowchart



Peripheral Event System is enabled, allowing the interconnection between one generator and two users:

- AST (Asynchronous Timer) as the generator
- CATB (Capacitive Touch Module) as a user
- ADCIFE (Analog to digital Converter) as a user

As described in the application flowchart (Figure 2-5), after the application initialization function, the user will either use a push button or a QTouch button to change the state of the sequential state machine implemented in the code.

2.2.2.1 ACTIVE_MODE

This state is the RUN mode of the application where:

- The main clock is 48MHz
- The LCD display is ON with the clock time displayed on it
- We stay in this mode while an external interrupt event (EIC controller) is not detected (PB0 pushed)

2.2.2.2 INIT_SLEEPWALKING

This state allows preparing the SAM4L device to go back to Wait mode, and configure peripherals to perform SleepWalking by:

- Disable PDCA used to manage transfer data to the LCD while the LCD is OFF to avoid extra power consumption in SleepWalking
- Restore the slow clock running @12MHz with FastRC as clock source (PS1 available)
- Enable the CATB module clock to allow QTouch interrupt
- Configure the ADC
- Initialize the AST
- Enable the Peripheral Event Controller to interconnect AST, CATB and the ADC
- Disable the LCD clock and the LCD Back Light to avoid extra power consumption in SleepWalking
- Disable the external the External Interrupt Controller (EIC) to avoid interrupt coming from PB0
- Change the State machine state to go into SLEEPWALKING

2.2.2.3 SLEEPWALKING

In this mode, the CPU clock is OFF, and only the 32kHz oscillator is enabled for the AST trig event activity. The PEVC is the peripheral which makes the SleepWalking available by interconnecting The AST as Generator and the CATB as user.

2.2.2.4 INIT_ACTIVE_MODE

Once in the state machine, this state is an intermediary state which is used just after wake up (touch on CS0) and allows configuring the peripheral used in ACTIVE state such as:

- Switch the clock to full speed to have a powerful data processing
- Disable the CATB module clock
- Enable PDCA used to manage transfer data to the LCD
- Enable and Initializes the LCD controller
- Enable the External Interrupt Controller (EIC)
- Enable LCD Back Light to display the time
- Change the State machine state to go into ACTIVE_MODE

2.3 Software Package Description

According to the “black box” mentioned previously, to correctly follow this application you have to download the related package which includes the code used to implement the real time watch application.

You will find the files:

- sleepwalking_appnote.c
- sleepwalking_appnote.h

These files have to be added to your project, once it has been created. You will be able to find all the functions used to realize the real time watch and to build therefore the application.



INFO

A more detailed description of these files is provided in annexes of this application note.

2.4 Main Steps Summary

Here are the outlines which describe how a SleepWalking application will be implemented:

Once familiarized with the SAM4L SleepWalking dedicated feature, the first steps will be to prepare correctly the project in Atmel Studio 6.

Then the SleepWalking implementation will begin by:

- Setup an Atmel Studio 6 Project:
 - Create an Atmel Studio 6 Project using a specific board template

- Add the sleepwalking_appnote files (.c and .h)
 - Import the related modules used in the application by using the ASF wizard
- Configure the Clocks and Low Power modes
 - Configure the main System Clock setting to use the PLL0 using the main oscillator as clock source to have a 48MHz running frequency
 - Configure the main System Clock setting to be able to jump in Wait mode
 - RCFast oscillator running at 12MHz
 - Allowing Fast wake up
 - Power Scaling set to PS1
- Configure the Peripherals
 - Configure the Asynchronous Timer (AST) to be able to trigger for the other peripherals
 - CATB and QTouch Library
 - Configure the ADCIFE with the Light Sensor
 - Configure the Peripheral Event System Controller (PEVC) to interconnect the AST trigger event to the CATB autonomous QTouch sensing, or to the ADCIFE
 - Configuring the External interrupt Controller (EIC) to go back to SleepWalking
- The last step will be to implement the State machine described in the [Figure 2-5](#)
- Compile and run the application with AST as generator and CATB and ADCIFE as users

3 Atmel Studio 6 Project Creation

3.1 Atmel Studio Introduction



Atmel Studio 6 is the integrated development platform (IDP) for developing and debugging Atmel ARM Cortex-M processor-based and Atmel AVR microcontroller applications.

The 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.

Download link: http://www.atmel.com/microsite/atmel_studio6/

Additionally, the IDP now includes two new features designed to further enhance your productivity: Atmel Gallery and Atmel Spaces:

- Atmel Gallery is an online apps store built in to Studio 6, allowing you to purchase both in-house and third-party development tools and embedded software.

Link: <http://gallery.atmel.com/>

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

Link: <http://spaces.atmel.com/gf/>

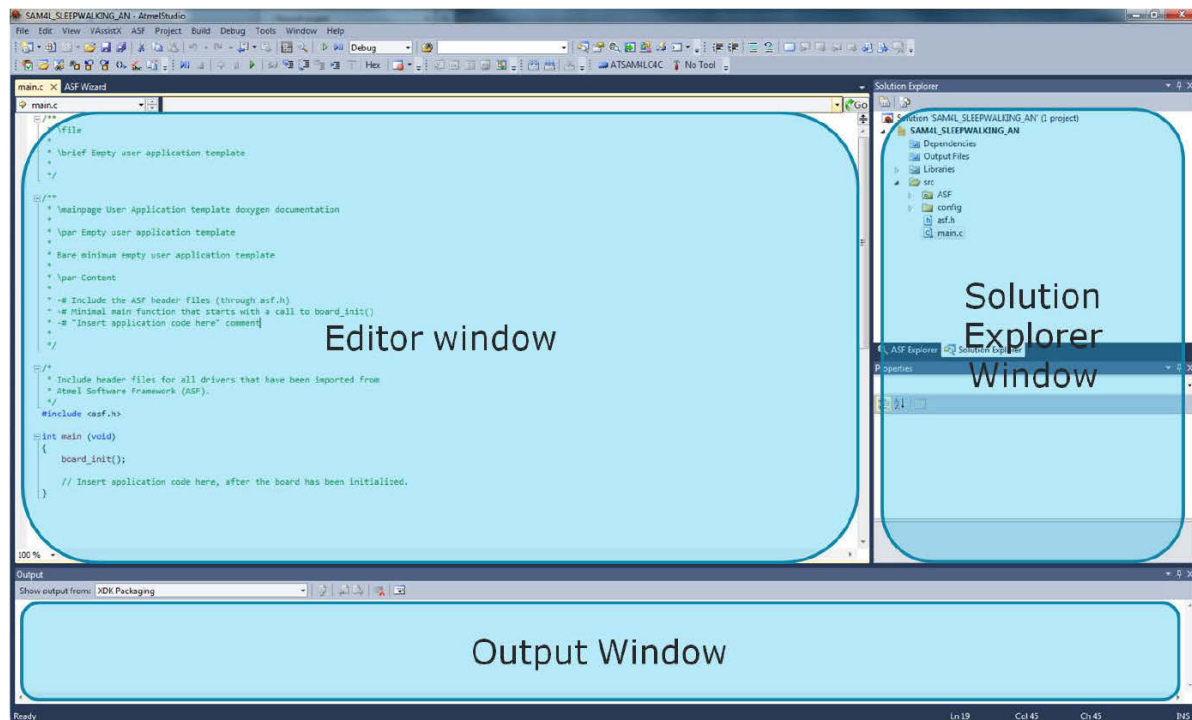


3.1.1 IDE Introduction

As described in the [Figure 3-1](#), the Atmel Studio 6 environment is composed of:

- An editor window, this is where the user will implement his code lines
- A Solution Explorer window, used to browse all your project directories
- An output window to read messages from the compiler

Figure 3-1. Atmel Studio 6.1 Overview



3.1.2 Atmel Software Framework (ASF) Introduction

The Atmel Software Framework (ASF) is a collection of embedded software for the Atmel Flash MCUs: megaAVR®, AVR XMEGA®, AVR UC3, and ARM Cortex-M processor-based devices.

It simplifies the use of our microcontrollers by providing an abstraction to the hardware and high-value middleware. ASF is designed to be used for evaluation, prototyping, design and production phases.

ASF is integrated in the Atmel Studio IDP with a graphical user interface or available as standalone for GCC, IAR™ compilers. ASF can be downloaded for free.

ASF Standalone for GCC and IAR link: <http://www.atmel.com/tools/AVRSOFTWAREFRAMEWORK.aspx>

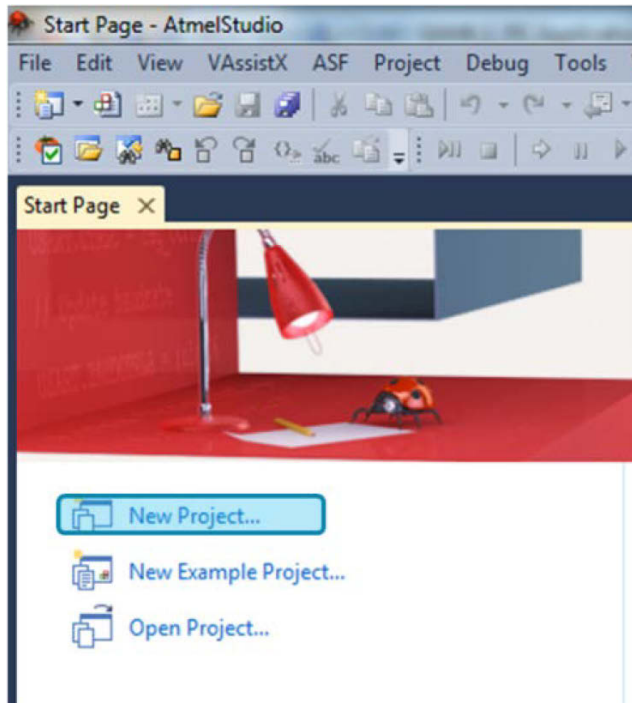
ASF Documentation: <http://asf.atmel.com/docs/latest/>

This example has been developed with the ASF version 3.13.1.

3.2 Create a Project under Atmel Studio using a Specific Board Template

After having launched Atmel Studio 6, the Start Page appears. To create a project using a specific board template, the “create project” menu must be chosen as described in [Figure 3-2](#).

Figure 3-2. Create a new Project from the Atmel Studio 6.1 Start Page

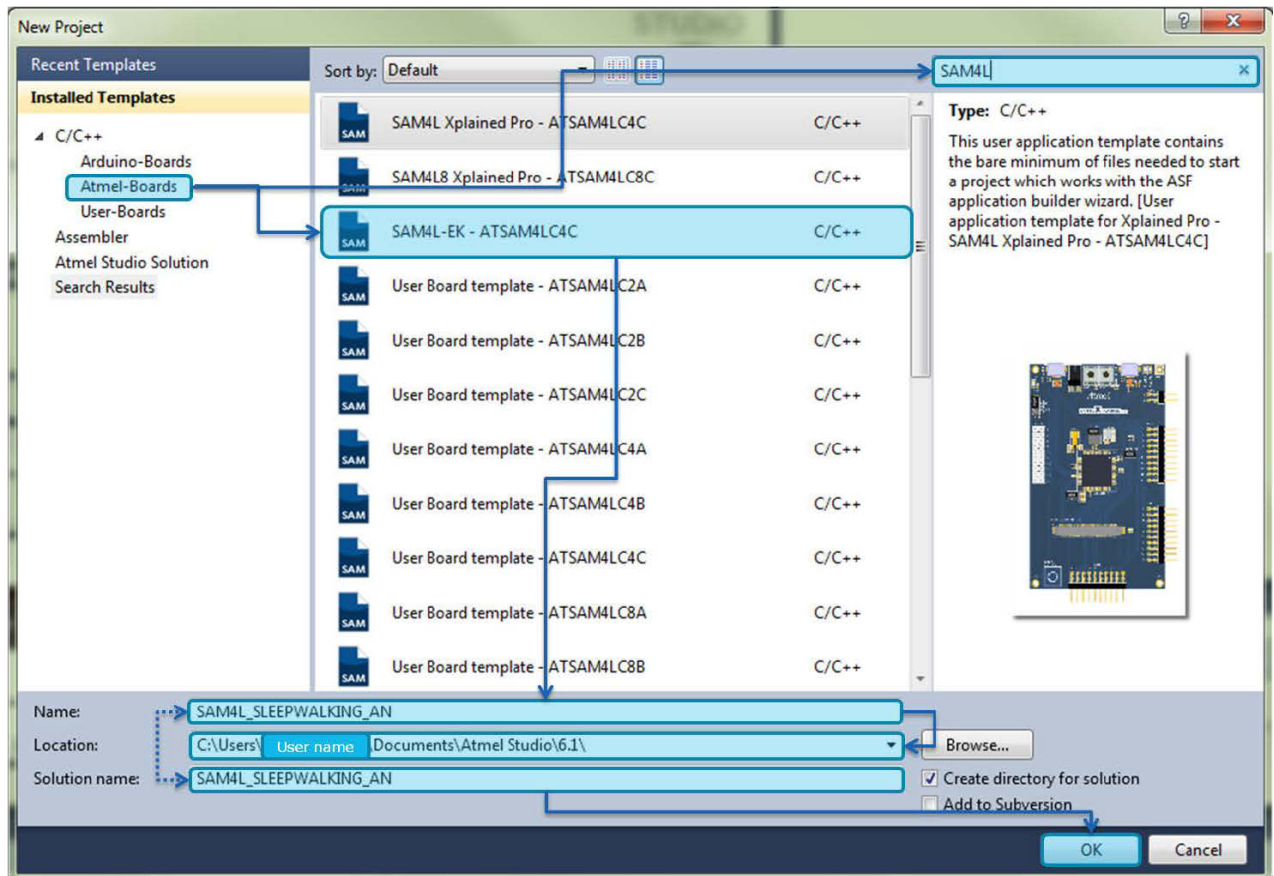


As shown in [Figure 3-3](#), the new project window appears. The SAM4L-EK board template has to be selected from the “Atmel-Boards” list which is selected in the “Installed Templates” from the left column. This will avoid all the board descriptions and initialization code writing. All these functions are already implemented in the Atmel Software Framework (ASF); it would be a mess to not keep this benefit. The project name and its related location are set from this menu as well.

In this example the following naming is used:

Project Name→	SAM4L_SLEEPWALKING_AN (Application Note)
Location→	C:\\Users\\username\\Documents\\atmel Studio\\6.1\\ (Default location)
Solution Name→	SAM4L_SLEEPWALKING_AN

Figure 3-3. SAM4L-EK Board Template Selection



Once these steps are done, the project is then generated and the main window appears as previously described in the [Figure 3-1](#).

At this time:

- A dedicated folder, called "SAM4L_SLEEPWALKING_AN", is generated at the project location "C:\Users\username\Documents\Atmel Studio\6.1". This folder contains the project and all the source files. All these files can be also opened from the solution explorer window (see [Figure 3-1](#)).
- The *main.c* file is automatically opened in the editor window and contains the following code lines:

```
/**
 * \file
 *
 * \brief Empty user application template
 *
 */

/**
 * \mainpage User Application template doxygen documentation
 *
 * \par Empty user application template
 *
 * Bare minimum empty user application template
 *
 * \par Content
```

```

*
* -# Include the ASF header files (through asf.h)
* -# Minimal main function that starts with a call to board_init()
* -# "Insert application code here" comment
*
*/

/*
* Include header files for all drivers that have been imported from
* Atmel Software Framework (ASF).
*/
#include <asf.h>

int main (void)
{
    board_init();

    // Insert application code here, after the board has been initialized.
}

```

- “`#include <asf.h>`” means that the project is based on the ASF library. The *asf.h* file contains all the header files related to the peripheral modules included into the project. When a project is created from a board template, the *asf.h* file already contains the minimum required modules to set main clocks, and to initialize the board, such as start up code, low level init, ioport, board description. This file is automatically updated when a new module is added to the project. Refer to [Section 3.4 Atmel Software Framework Module Importation](#).
- “*board_init()*,” refers to *init.c* file located under the following ASF folder.
`SAM4L_SLEEWALKING_AN\src\ASF\sam\boards\sam4l_ek\`. This file allows a first configuration of the board and complies with the ASF board description available at the same location: the *sam4l_ek.h* header file.

3.3 Add Existing File to the Project

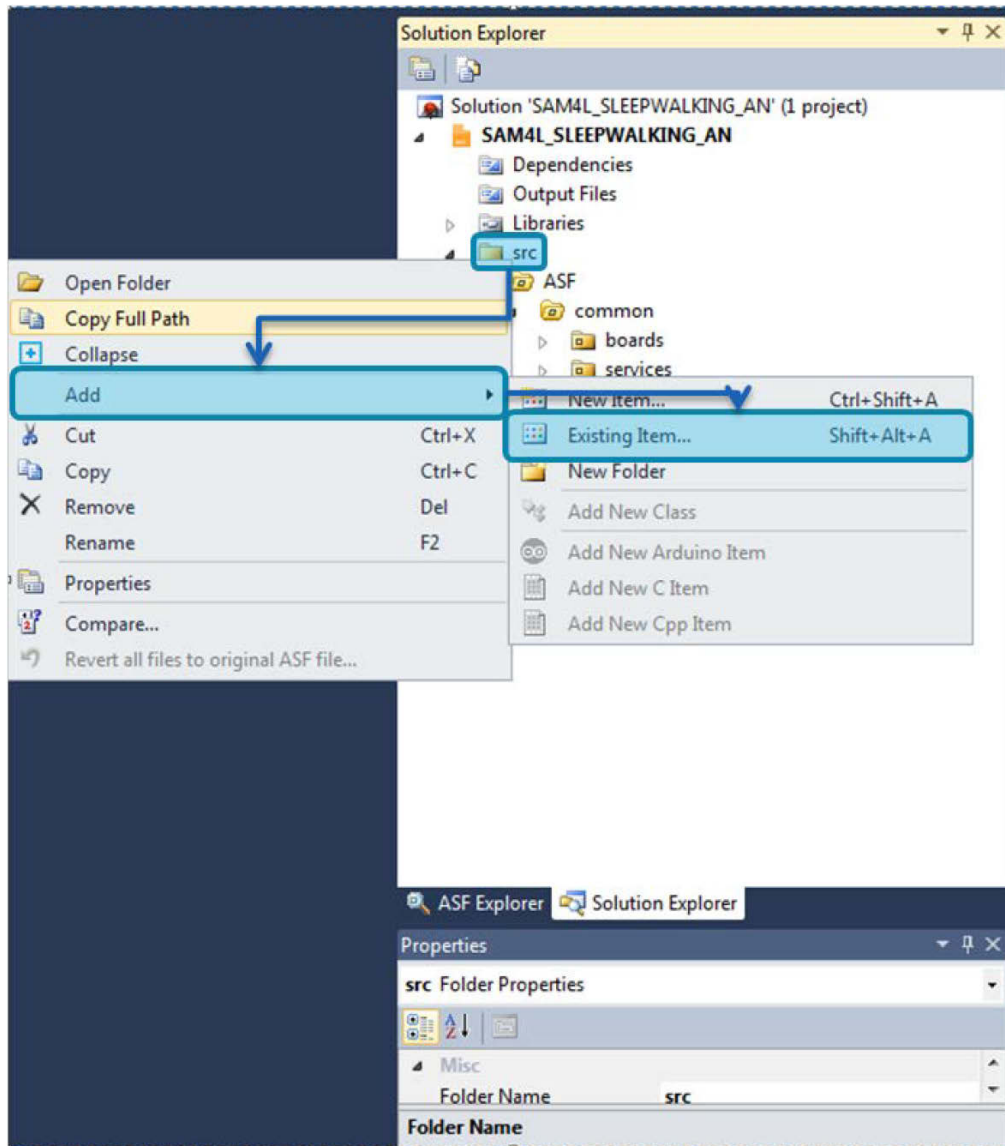
Once the project is created, the *sleepwalking_apnote* files (.c & .h) have to be added to the project. These files are initially located into the Application note package, under the following path:

`SAM4L_EK_Application_Note_SleepWalking_ADCIFE_&_Qtouch_rev_0.1\ATMEL_STUDIO_PROJECT\Solution\example library`.

To add these files to the project:

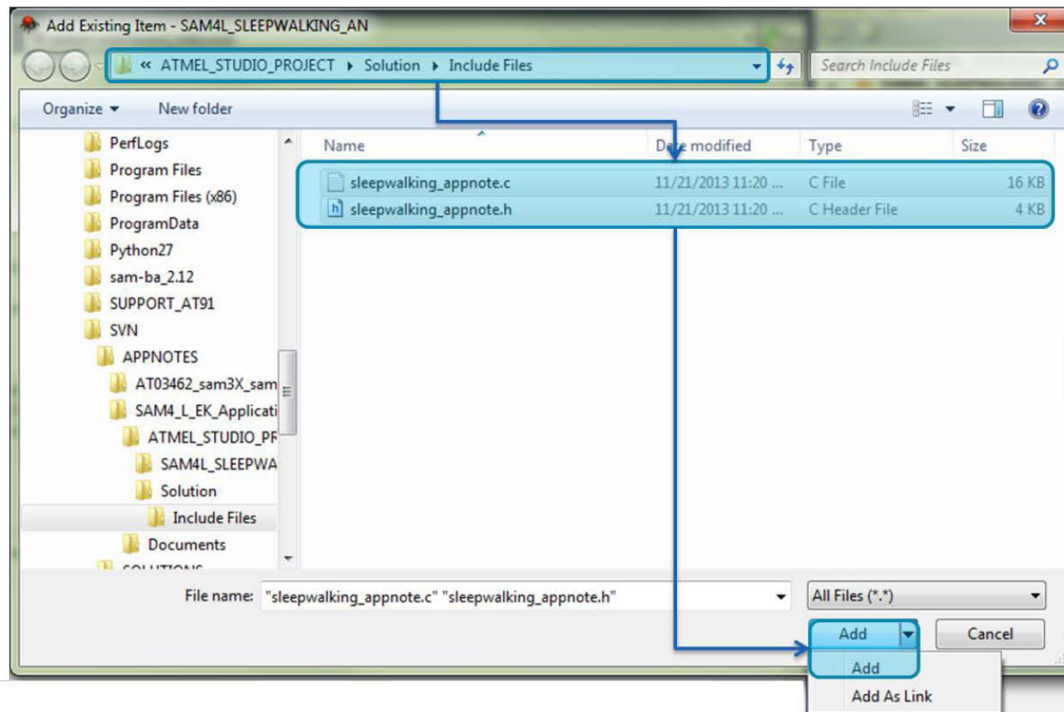
- From the solution explorer window:
 - **Right click** on the targeted destination folder, “*src*”
 - **Select** the “*Add*” menu
 - **Select** “*Existing Item...*” as described:

Figure 3-4. Access to the Add File Menu



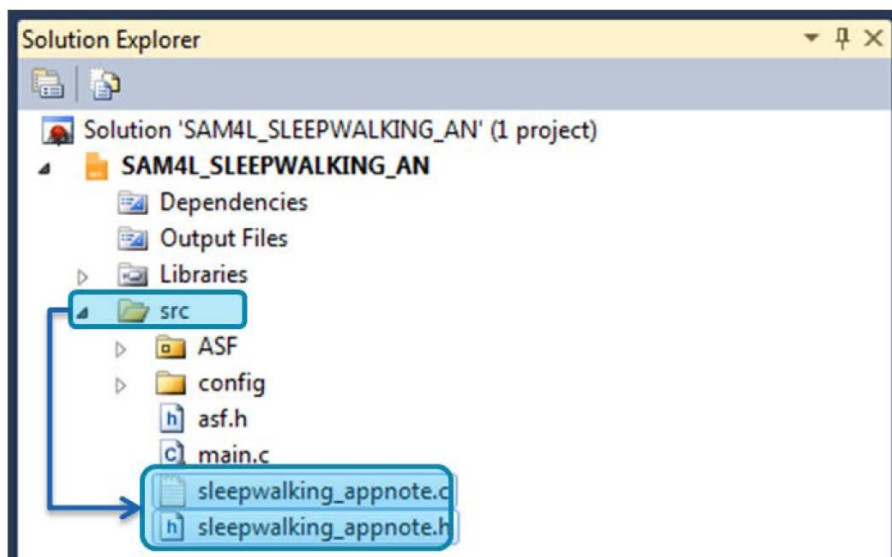
- **Browse the explorer and go** to the path previously mentioned
“SAM4_L_EK_Application_Note_SleepWalking_ADCIFE_&_Qtouch_rev_0.1\ATMEL_STUDIO_PROJ\CT\Solution\example library”
 - **Select** the two files with CTRL
 - **Click** on “Add”. The files **must be** copied into your src project folder. That means that a particular care has to be taken regarding the add button. To do that, the “Add” option must be chosen, and not the “Add as a link” option, which will not copy the file into the src directory.

Figure 3-5. Add the sleepwalking_appnote .c and .h Files



Once the files are added to the project, they should appear in the *src* folder from the *Solution Explorer* window like this:

Figure 3-6. The sleepwalking_appnote .c and .h Files once added to the Atmel Studio 6 Project



3.4 Atmel Software Framework Module Importation

In this example several peripherals, services, drivers and components are used to implement the whole SleepWalking application.

As a first task just after the project creation, is to add the related ASF modules.

To open the ASF wizard:


- **Click** on the button  from the Standard Tool Bar. The ASF Wizard window tab appears from the editor window area:

Figure 3-7. ASF Wizard Window Tab

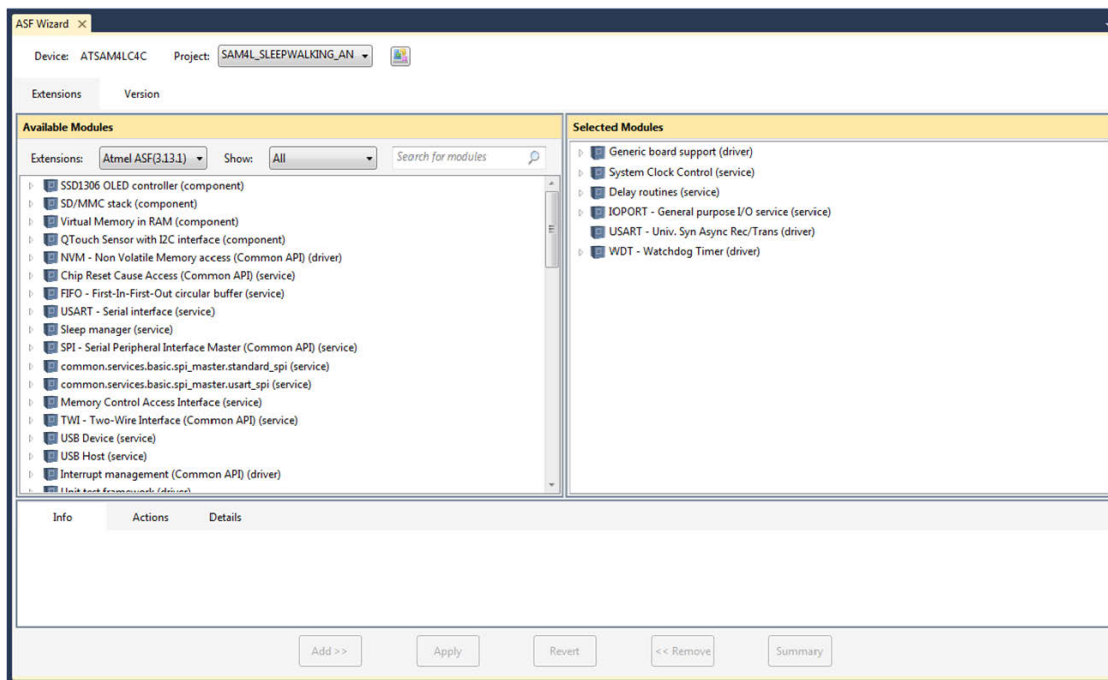


Figure 3-8 and Figure 3-9 describe the list of modules which are included by default for a new project creation, and the one which have to be added for this example of application.

Figure 3-8. The ASF Selected Modules List at the End of the Modules Importation

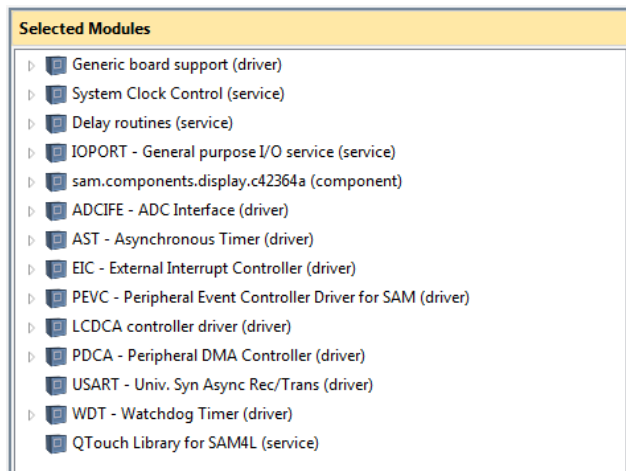
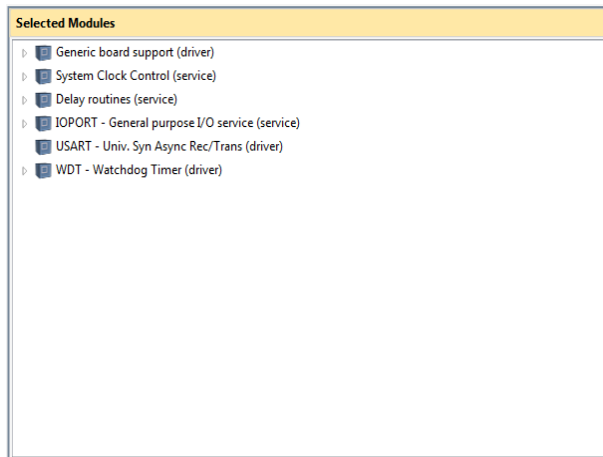


Figure 3-9. The ASF Selected Modules List at the Project Creation



As described in Chapter 2 [SleepWalking Application Description](#), several peripherals will be used in this application. Therefore the following modules have to be imported:

- **ADCIFE - ADC Interface.** Driver for the ADCIFE (Analog-to-Digital Converter Interface). Provides functions for configuration of conversion parameters (up to 12-bit signed at 350ksps), channel sequencing (max. 16 channels, w/ 16 different inputs and up to 64x gain), window monitoring, interrupt and conversion triggering
- **AST - Asynchronous Timer.** Driver for the AST: Provides functions for configuring and operating the AST in the calendar or timer/counter modes. Or to configure the Asynchronous Timer (AST) to be able to trigger event for the other peripherals
- **EIC - External Interrupt Controller.** Provides functions for EIC allowing pins to be configured as external interrupts. As for example to configure the External interrupt Controller (EIC) to go back to SleepWalking
- **PEVC - Peripheral Event Controller** Driver for SAM. Provide a unified interface for the configuration and management of the event channels within the device. For example to configure the Peripheral Event System Controller (PEVC) to interconnect the AST trigger event to the CATB autonomous QTouch sensing, or to the ADCIFE
- **QTouch Library for SAM4L:** CATB and QTouch Library
- Exclusively used for the Black Box Application:
 - **PDCA - Peripheral DMA Controller.** The Peripheral DMA Controller transfers data between on-chip serial peripherals and the on- and/or off-chip memories. The link between the PDC and a serial peripheral is operated by the AHB to APB bridge
 - **LCDCA Controller driver.** Driver for the LCDCA controller. Provides functions for using the on-chip LCDCA controller
 - **Sam.components.display.c42364a (component):** Low-level driver for the C42364 LCD Glass

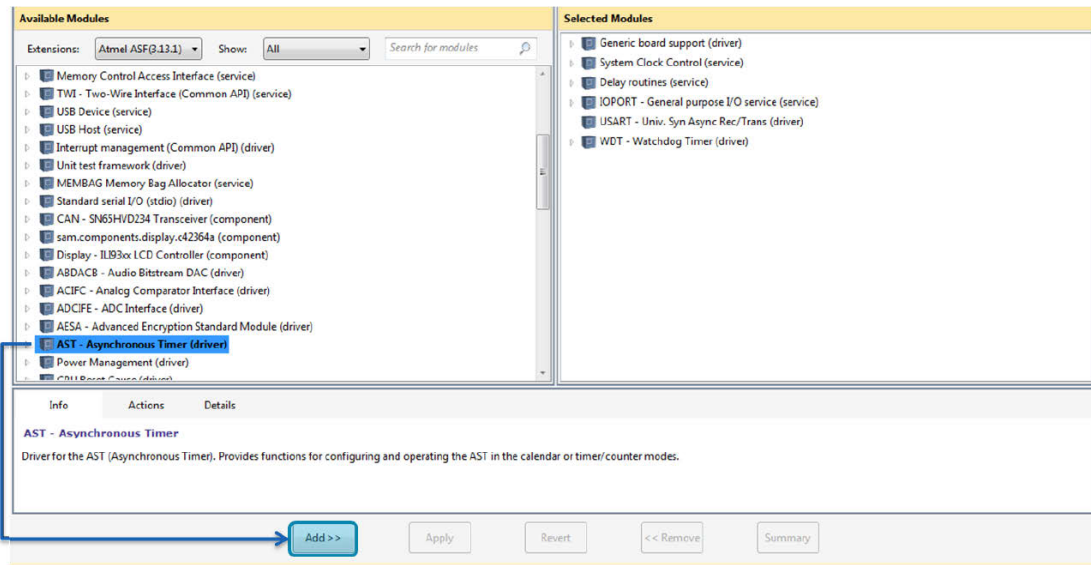
To import a module into a new project is a very simple task in Atmel Studio 6. Follow the instructions below to add a module and repeat them again for all the modules to be added into the project.

Here is a module importation example:

To add the AST (Asynchronous Timer) Module into the Atmel Studio Project:

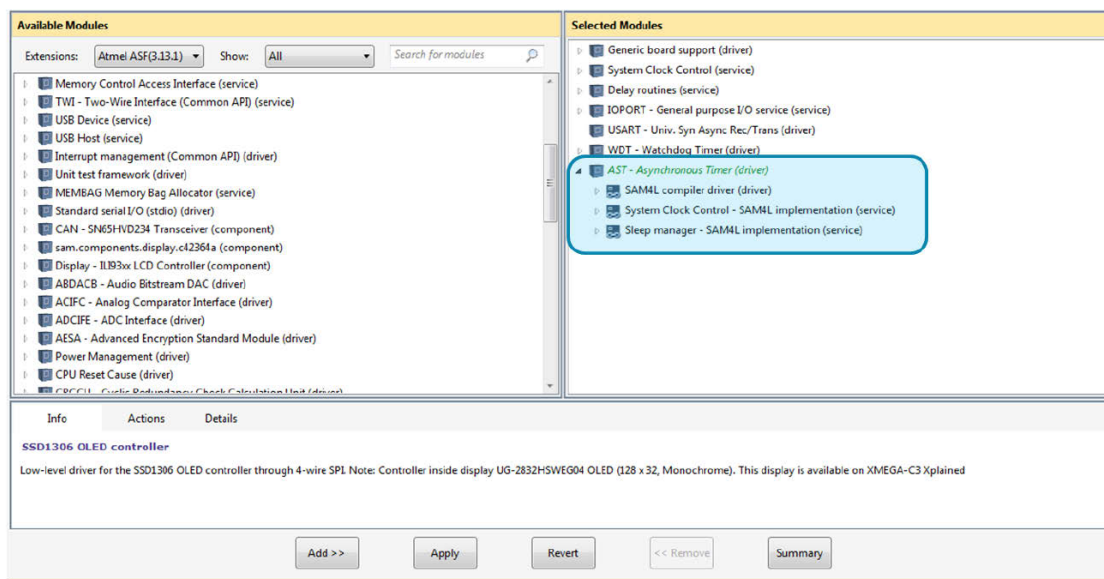
- **Select** from the “Available Modules” list the *AST - Asynchronous Timer (driver)* item

Figure 3-10. Module Selection



- **Click** on the “Add” button

Figure 3-11. Module Addition



- **Repeat** these last two steps (Select and Add) for the ADCIFE, EIC, PEVC, the QTouch Library, and the LCD Glass Drivers
- Finally **click** on the “Apply” button to apply your modules importation to your project

Figure 3-12. Module Addition

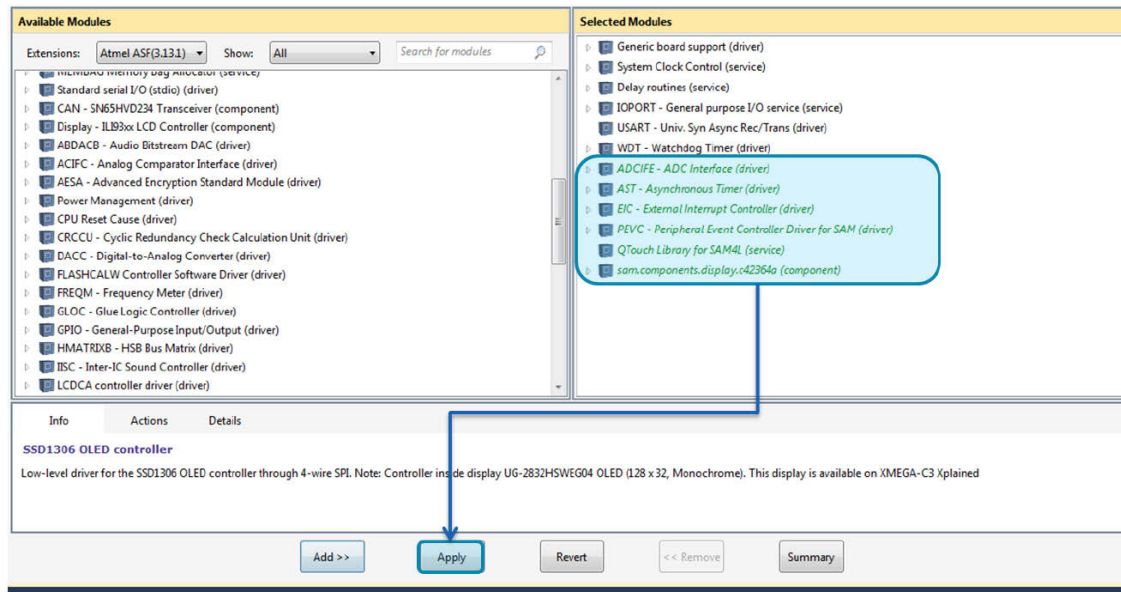
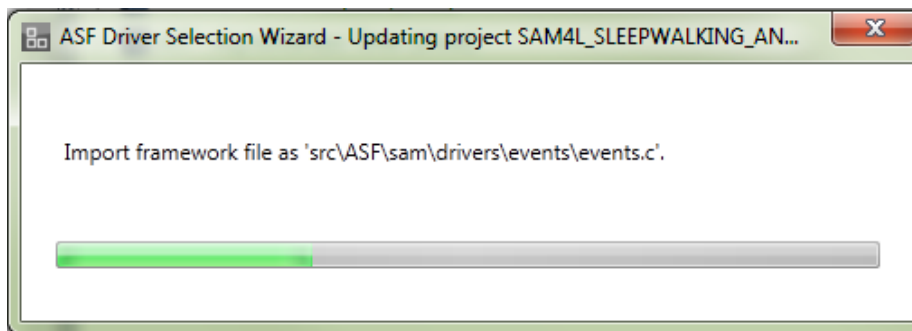


Figure 3-13. Module Addition Processing

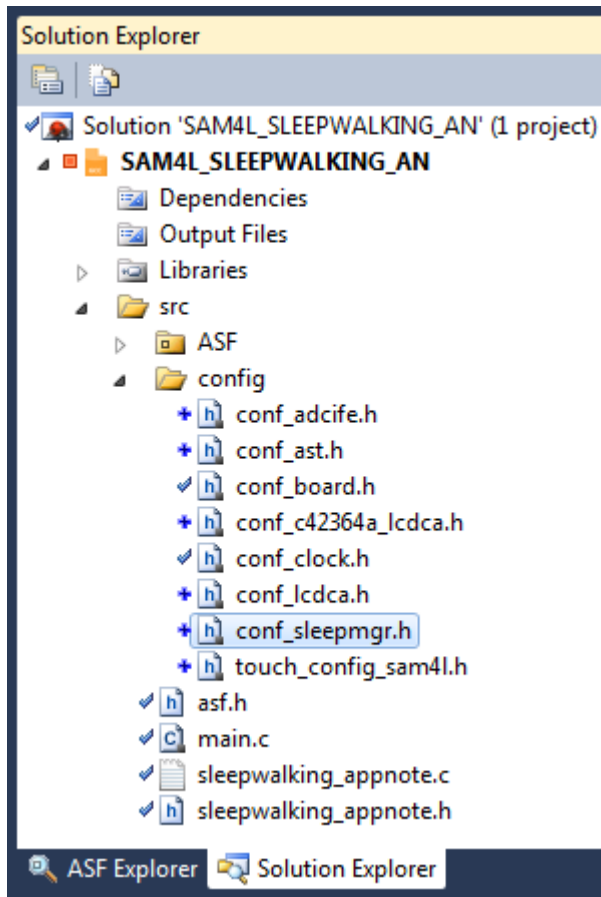


Once the importation process has finished, the Atmel Studio 6 project should contain all the driver files related to the modules which just has been added. The asf.h file is also automatically modified accordingly. The table below just sums up the main resulting project modifications after the modules importation:

Modules	Driver paths	files	Config. files
ADCIFE	SAM4L_SLEEPWALKING_AN\src\ASF\sam\drivers\adcife\	adcife.c adcife.h	conf_adcife.h
AST	SAM4L_SLEEPWALKING_AN\src\ASF\sam\drivers\ast\	ast.c ast.h	conf_ast.h
EIC	SAM4L_SLEEPWALKING_AN\src\ASF\sam\drivers\aic\	eic.c eic.h	conf_lcdca.h
PEVC	SAM4L_SLEEPWALKING_AN\src\ASF\sam\drivers\lbp\	bpm.c bpm.h	
	SAM4L_SLEEPWALKING_AN\src\ASF\sam\drivers\flashcalw\	flashcalw.c flashcalw.h	
QTouch	SAM4L_SLEEPWALKING_AN\src\ASF\thirdparty\qtouch\devspecific\sam4\sam4\nlib\gcc\	libsam4l-qt-gnu.a	touch_config_sam4l.h
	SAM4L_SLEEPWALKING_AN\src\ASF\thirdparty\qtouch\devspecific\sam4\sam4\ninclude\	touch_api_sam4l.h	
LCD	SAM4L_SLEEPWALKING_AN\src\ASF\sam\drivers\lcdca\	lcdca.c lcdca.h	conf_lcdca.h conf_c42364a_lcdca.h

All the configuration files should be added to the project in the `SAM4L_SLEEPWALKING_AN\SAM4L_SLEEPWALKING_AN\src\config\` folder, as described below:

Figure 3-14. The Configuration Folder

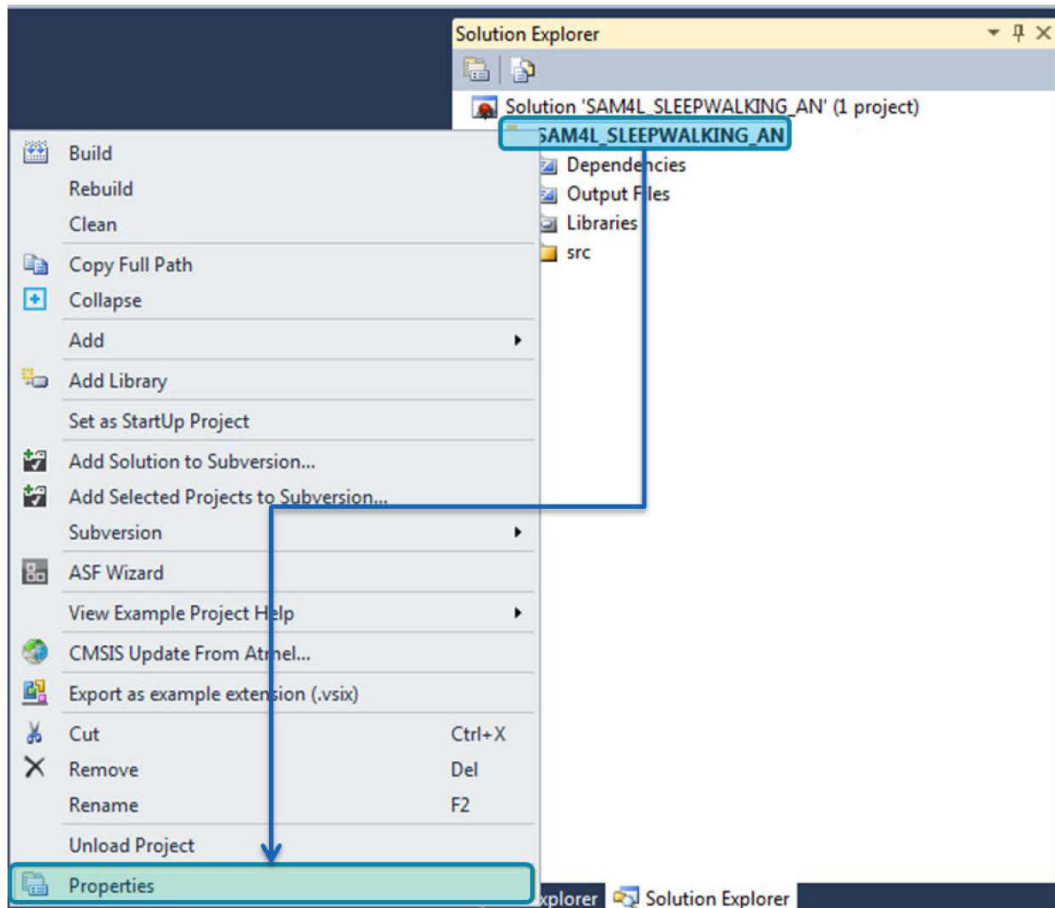


3.5 Debugger Settings

The SAM4L-EK board is designed to use the Serial Wire Debug protocol (**SWD**) instead of standard JTAG. Thus, the project has to be correctly configured to use the SWD interface in order to be able to program the chip. To configure the project to use the SWD protocol instead of JTAG, the project properties have to be modified as follow:

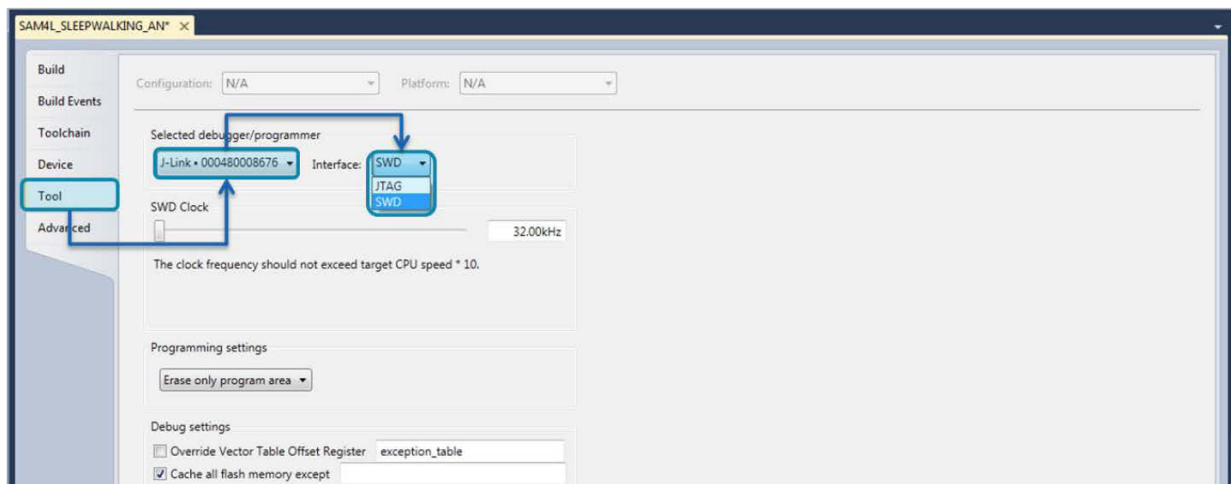
- As described in [Figure 3-13](#), from the solution explorer window, **right click** on the `SAM4L_SLEEPWALKING_AN` project file name and **select** “*Properties*”


Figure 3-15. Open Project Properties in Atmel Studio 6.1



- **Select then, as the Figure 3-14 shows:**
 - Tool Tab
 - Then, J-Link- debugger/programmer (J-Link
 - Finally the Serial Wire Interface (SWD)

Figure 3-16. Change the Debugger/programmer Interface in Atmel Studio 6.1



- **Save** the new project properties to apply them simply by **clicking** on the save button: 

4 System Initialization

In this chapter the code implementation will follow the functional diagram described by the [Figure 2-6 App_init Flowchart](#) in the [Section 2.2.2 Function Descriptions](#).

4.1 Starting Point

As a starting point, the *main.c* file looks like this:

```
#include <asf.h>

int main (void)
{
    board_init();

    // Insert application code here, after the board has been initialized.
}
```

Before entering in the *main* function the program will start by the low level initialization, which is implemented in the *startup_sam4l.c* located in the following path:

SAM4L_SLEEPWALKING_AN\src\ASF\sam\utils\cmsis\sam4l\source\templates\gcc.

At this step, the *board_init()* function is the only content of the main function. Its aim consists into configuring the IOPORTs of the chip according to the SAM4L-EK hardware components. This function is implemented in the *init.c* file located in the folder *SAM4L_SLEEPWALKING_AN\src\ASF\sam\boards\sam4l_ek*.

As a preliminary step, we have to include to the *main.c* file the example library files.

4.1.1 Include Example Library File

The *sleepwalking_appnote.c* and *.h* files contain the “Black Box” application code and they have to be included from the *main.c* file as described:



INFO

In the whole document, all the modifications are highlighted in **BOLD**.

```
/*
 * Include header files for all drivers that have been imported from
 * Atmel Software Framework (ASF).
 */
#include <asf.h>

/* Include sleepwalking_appnote header */
#include "sleepwalking_appnote.h"

int main (void)
{
    board_init();

    // Insert application code here, after the board has been initialized.
}
```

4.2 Application Init Function Implementation

All the initialization functions are grouped into one *app_init()* function (refer to [Figure 2-6](#)). Therefore the main function can be directly modified as follow:


```

/*
 * Include header files for all drivers that have been imported from
 * Atmel Software Framework (ASF).
 */
#include <asf.h>

/* Include sleepwalking_appnote header */
#include "sleepwalking_appnote.h"

/**
 * Application Initialization function Implementation
 */
static void app_init(void)
{
    /* Initialize the SAM4L-EK board */
    board_init();
} //end app_init

/**
 * MAIN Function
 */
int main (void)
{
    app_init();

    // Insert application code here, after the board has been initialized.
}

```

As described above, the *app_init* function has been implemented just before the *main* function, and the *board_init* function moved from the main function to the *app_init* function.



INFO

In the next coming paragraph the focus will be done on the *app_init* function implementation to reach the model described by [Figure 2-6](#).

4.3 Application's Clock Setting Configuration

When ASF is used, the clock settings functions are already provided thanks to the *System Clock Control* module service. This module has the following implementation:

Modules	Driver paths	files	Config files
Clocks	SAM4L_SLEEPWALKING_AN\src\ASF\common\services\clock\sam4l\	sysclk.c sysclk.h pll.c pll.h osc.c osc.h genclk.h dfll.c dfll.h	conf_clock.h

As described above, each couple of .c and .h files relates to a SAM4L dedicated clock source driver, except the *sysclk.c* and .h files which implement functions configuring the clock sources according to the *conf_clock.h* located in the folder *SAM4L_SLEEPWALKING_AN\src\config* which is described below:

```
/**
 * \file
 *
 * \brief Chip-specific system clock manager configuration
 *
 */
#ifndef CONF_CLOCK_H_INCLUDED
#define CONF_CLOCK_H_INCLUDED

// #define CONFIG_SYSCLK_INIT_CPUMASK (1 << SYSCLK_OCD)
// #define CONFIG_SYSCLK_INIT_PBAMASK (1 << SYSCLK_IISC)
// #define CONFIG_SYSCLK_INIT_PBBMASK (1 << SYSCLK_USBC_REGS)
// #define CONFIG_SYSCLK_INIT_PBCMASK (1 << SYSCLK_CHIPID)
// #define CONFIG_SYSCLK_INIT_PBDMASK (1 << SYSCLK_AST)
// #define CONFIG_SYSCLK_INIT_HSBMASK (1 << SYSCLK_PDCA_HSB)

#define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_RCSYS
// #define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_OSC0
// #define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_PLL0
// #define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_DPLL
// #define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_RC80M
// #define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_RCFAST
// #define CONFIG_SYSCLK_SOURCE        SYSCLK_SRC_RC1M

/* RCFAST frequency selection: 0 for 4MHz, 1 for 8MHz and 2 for 12MHz */
// #define CONFIG_RCFAST_FRANGE        0
// #define CONFIG_RCFAST_FRANGE        1
// #define CONFIG_RCFAST_FRANGE        2

/* 0: disable PicoCache, 1: enable PicoCache */
#define CONFIG_HCACHE_ENABLE        1

/*
 * To use low power mode for flash read mode (PS0, PS1), don't define it.
 * To use high speed mode for flash read mode (PS2), define it.
 *
 * \note
 * For early engineer samples, ONLY low power mode support for flash read mode.
 */
// #define CONFIG_FLASH_READ_MODE_HIGH_SPEED_ENABLE

/* Fbus = Fsys / (2 ^ BUS_div) */
#define CONFIG_SYSCLK_CPU_DIV        0
#define CONFIG_SYSCLK_PBA_DIV        0
#define CONFIG_SYSCLK_PBB_DIV        0
#define CONFIG_SYSCLK_PBC_DIV        0
#define CONFIG_SYSCLK_PBD_DIV        0
```

```

// #define CONFIG_USBCLK_SOURCE          USBCLK_SRC_OSC0
#define CONFIG_USBCLK_SOURCE          USBCLK_SRC_PLL0
// #define CONFIG_USBCLK_STARTUP_TIMEOUT (OSC0_STARTUP_TIMEOUT*(1000000/OSC_RCSYS_NOMINAL_HZ))

/* Fusb = Fsys / USB_div */
#define CONFIG_USBCLK_DIV              1

#define CONFIG_PLL0_SOURCE             PLL_SRC_OSC0

/* Fpll0 = (Fclk * PLL_mul) / PLL_div */
#define CONFIG_PLL0_MUL                (48000000UL / BOARD_OSC0_HZ)
#define CONFIG_PLL0_DIV                1

// #define CONFIG_DPLL0_SOURCE          GENCLK_SRC_RCSYS
// #define CONFIG_DPLL0_SOURCE          GENCLK_SRC_OSC32K
// #define CONFIG_DPLL0_SOURCE          GENCLK_SRC_RC32K

/* Fdpll = (Fclk * DPLL_mul) / DPLL_div */
// #define CONFIG_DPLL0_FREQ            48000000UL
// #define CONFIG_DPLL0_MUL              (CONFIG_DPLL0_FREQ / BOARD_OSC32_HZ)
// #define CONFIG_DPLL0_DIV              1

#endif /* CONF_CLOCK_H_INCLUDED */

```

As shown from this file above, the clock configuration is determined by the definition settings chosen by the user. As explained by the [Figure 2-5](#) and then by the [Section 2.2.2.1 ACTIVE_MODE](#), just after the initialization process, the application will jump directly into the Active mode of the state machine.

Therefore, the parameters of the *conf_clock.h* file must be set to get:

- 48MHz core clock
- Using the on board crystal oscillator
- Using the PLL0 for the multiplication

To get this configuration the main parameters to modify are:

Parameter definition	Parameter setting definitions	Comments
CONFIG_SYSCLK_SOURCE	=SYSCLK_SRC_PLL0	System Clock source set to PLL0. This value is then set in the MCSEL bitfield of the MCCTRL register.
CONFIG_PLL0_SOURCE	=PLL_SRC_OSC0	PLL0 clock source is the main crystal oscillator (OSC0) running @ 12MHz. This value is then set into the PLL0SC bitfield of the PLL0 control Register. And the OSC0 is enabled (SCIF.OSCEN).
CONFIG_PLL0_MUL	=(48000000/BOARD_OSC0_HZ) =4	The multiply factor of the PLL0. This value is then set into the PLLMUL bitfield of the PLL0 control Register. Here the on board oscillator must multiplied by 4 (12MHz*4 = 48MHz).
CONFIG_PLL0_DIV	=1 (no division required)	The division factor of the PLL0. This value is then set into the PLLDIV bitfield of the PLL0 control Register.



INFO

All the parameter setting definitions are described in the *sysclk.h*, *osc.h*, and in the *pll.h* files.

Here are the **modifications** to be done from the current state of the *conf_clock.h* file:

```
/**
 * \file
 *
 * \brief Chip-specific system clock manager configuration
 *
 */
#ifndef CONF_CLOCK_H_INCLUDED
#define CONF_CLOCK_H_INCLUDED

// #define CONFIG_SYSCLK_INIT_CPUMASK (1 << SYSCLK_OCD)
// #define CONFIG_SYSCLK_INIT_PBAMASK (1 << SYSCLK_IISC)
// #define CONFIG_SYSCLK_INIT_PBBMASK (1 << SYSCLK_USBC_REGS)
// #define CONFIG_SYSCLK_INIT_PBCMASK (1 << SYSCLK_CHIPID)
// #define CONFIG_SYSCLK_INIT_PBDMASK (1 << SYSCLK_AST)
// #define CONFIG_SYSCLK_INIT_HSBMASK (1 << SYSCLK_PDCA_HSB)

// #define CONFIG_SYSCLK_SOURCE SYSCLK_SRC_RCSYS TO COMMENT
// #define CONFIG_SYSCLK_SOURCE SYSCLK_SRC_OSC0
#define CONFIG_SYSCLK_SOURCE SYSCLK_SRC_PLL0
// #define CONFIG_SYSCLK_SOURCE SYSCLK_SRC_DPLL
// #define CONFIG_SYSCLK_SOURCE SYSCLK_SRC_RC80M
// #define CONFIG_SYSCLK_SOURCE SYSCLK_SRC_RCFAST
// #define CONFIG_SYSCLK_SOURCE SYSCLK_SRC_RC1M

/* RCFAST frequency selection: 0 for 4MHz, 1 for 8MHz and 2 for 12MHz */
// #define CONFIG_RCFAST_FRANGE 0
// #define CONFIG_RCFAST_FRANGE 1
// #define CONFIG_RCFAST_FRANGE 2

/* 0: disable PicoCache, 1: enable PicoCache */
#define CONFIG_HCACHE_ENABLE 1

/*
 * To use low power mode for flash read mode (PS0, PS1), don't define it.
 * To use high speed mode for flash read mode (PS2), define it.
 *
 * \note
 * For early engineer samples, ONLY low power mode support for flash read mode.
 */
// #define CONFIG_FLASH_READ_MODE_HIGH_SPEED_ENABLE

/* Fbus = Fsys / (2 ^ BUS_div) */
#define CONFIG_SYSCLK_CPU_DIV 0
#define CONFIG_SYSCLK_PBA_DIV 0
#define CONFIG_SYSCLK_PBB_DIV 0
```

```

#define CONFIG_SYSCLK_PBC_DIV      0
#define CONFIG_SYSCLK_PBD_DIV      0

// #define CONFIG_USBCLK_SOURCE      USBCLK_SRC_OSC0
#define CONFIG_USBCLK_SOURCE      USBCLK_SRC_PLL0
// #define CONFIG_USBCLK_STARTUP_TIMEOUT (OSC0_STARTUP_TIMEOUT*(1000000/OSC_RCSYS_NOMINAL_HZ))

/* Fusb = Fsys / USB_div */
#define CONFIG_USBCLK_DIV          1

#define CONFIG_PLL0_SOURCE          PLL_SRC_OSC0

/* Fpll0 = (Fclk * PLL_mul) / PLL_div */
#define CONFIG_PLL0_MUL              (48000000UL / BOARD_OSC0_HZ)
#define CONFIG_PLL0_DIV              1

// #define CONFIG_DFLLO_SOURCE          GENCLK_SRC_RCSYS
// #define CONFIG_DFLLO_SOURCE          GENCLK_SRC_OSC32K
// #define CONFIG_DFLLO_SOURCE          GENCLK_SRC_RC32K

/* Fdfl1 = (Fclk * DFLL_mul) / DFLL_div */
// #define CONFIG_DFLLO_FREQ              48000000UL
// #define CONFIG_DFLLO_MUL              (CONFIG_DFLLO_FREQ / BOARD_OSC32_HZ)
// #define CONFIG_DFLLO_DIV              1

#endif /* CONF_CLOCK_H_INCLUDED */

```

At this time the *conf_clock.h* file is correctly modified according to the application requirements. But the *main.c* file must be modified by calling the *sysclk_init* function (implemented in the *sysclk.c* file) from the *app_init* function as described:

```

/**
 * Initialize the Application System and Peripherals for the example.
 */
static void app_init(void)
{
    /* Initialize the SAM4L-EK board */
    board_init();

    /* Application's clock setting configuration */
    sysclk_init();
} //end app_init

```

The *sysclk_init* function is the ASF function used to apply these modifications into the related registers (MCCTRL.MCSEL, SCIF.OSCEN etc...).

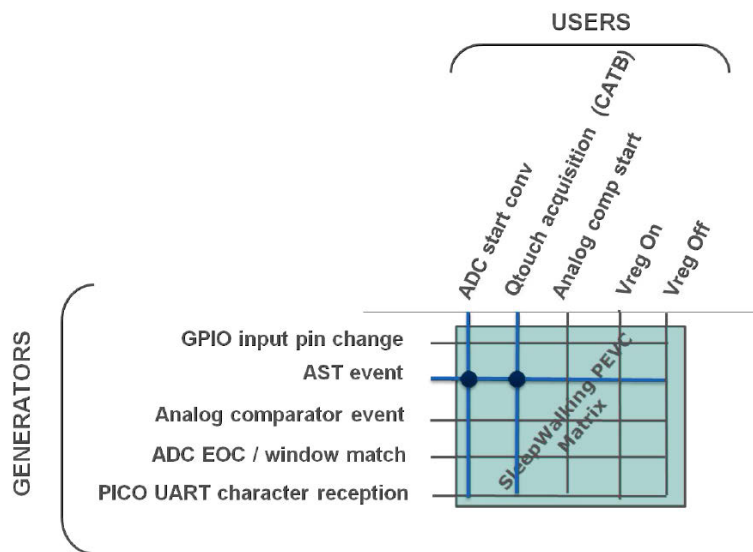
5 SAM4L Peripherals Configuration Method

5.1 Application Peripheral Configuration Overview

After having set the clock correctly, each peripheral configuration function can be implemented as described in [Figure 2-6 App_init Flowchart](#). The first function would be the *init_ADCIFE*, and the last would be the *init_AST*. The aim is to configure the peripheral registers getting:

- ADC and the CATB (touch) as peripheral users interconnected through the PEVC, to the AST which is a generator as described in [Figure 5-1](#)
- EIC to catch interrupts coming from PB0
- The “Black Box” application initialized (set the clock time)

Figure 5-1. PEV SleepWalking Matrix



5.2 Methodology

By using ASF, the methodology to configure a peripheral is really similar between each peripheral. Before starting to implement the peripheral configuration functions, let's try to summarize the step by step method:

1. Read main information at the peripheral chapter from product datasheet (for sure...).
2. Check if any hardware board configuration is required before peripheral using (Jumpers, PIO multiplexing, external components relationships).



INFO

These two first steps are not specific to ASF using...

3. Use the online API documentation which provides simple example on each ASF modules. This will explain how the driver files (.c and .h) (e.g.: *adc.h* and *adc.c*) are built to identify:
 1. What are the existing structures to be used to configure the peripheral?
 2. What the functions to be used as well..?
4. Declare the configuration structure type variables.
5. Get the default values of each structure field (if required).
6. Set The configuration Structure.
7. Apply the settings.

8. Enable the peripheral clocks (at bridge level and at cell level).
9. Enable interrupts (at core level and at peripheral level).

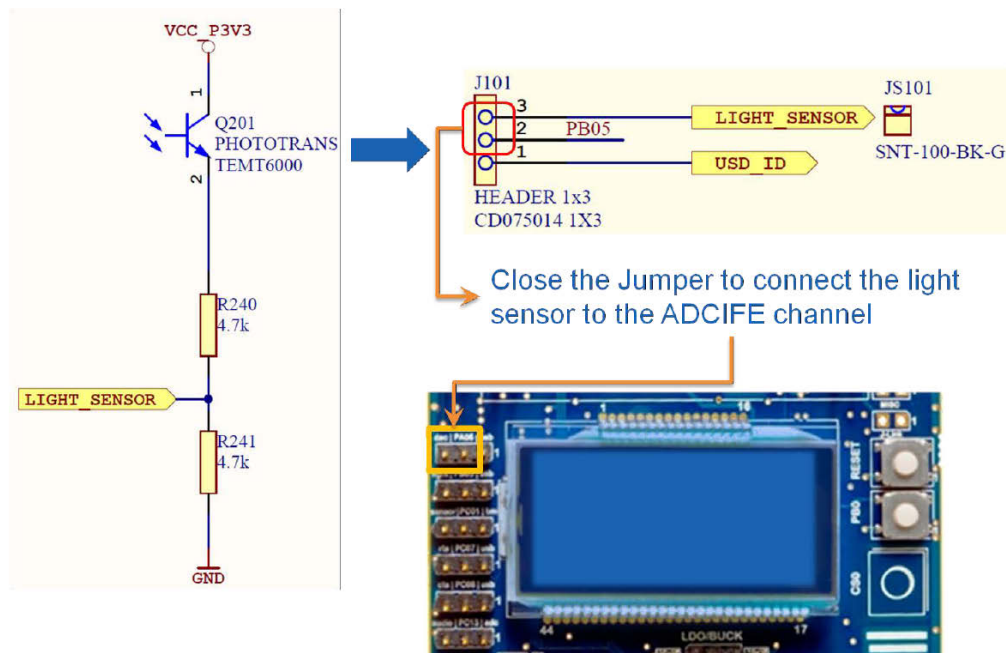
By assuming that the user has already performed Step 1 for the next coming paragraphs, the peripheral configurations should be really close to this methodology starting from Step 2.

6 Analog to Digital Converter (ADCIFE) Configuration

6.1 Check any Hardware Configuration Related to the Peripheral

The aim is to use the on board Light Sensor on the SAM4L-EK, which is directly connected to the *PB5 GPIO* as described by the schematic below: (refer to the SAM4L-EK design manual).

Figure 6-1. How to Connect the Embedded li

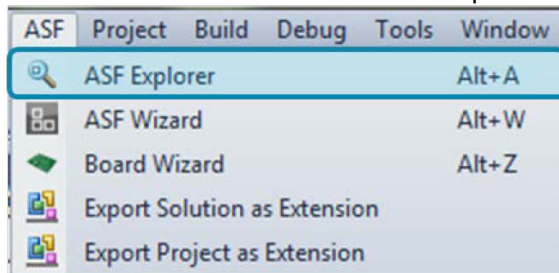


This is done by closing the J101 jumper in position 2-3.

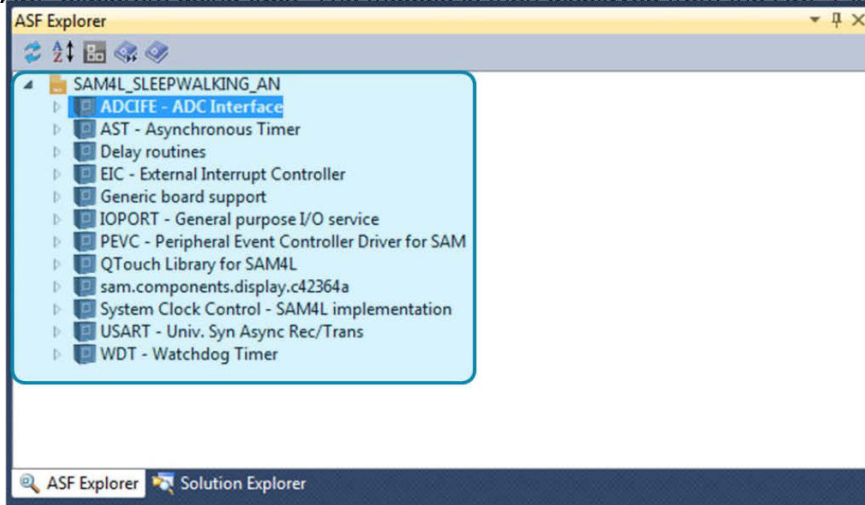
6.2 Use the Online API Documentation which provides Simple Example on ADCIF

To open the online documentation directly from Atmel Studio 6, make sure having the ASF Explorer window opened:

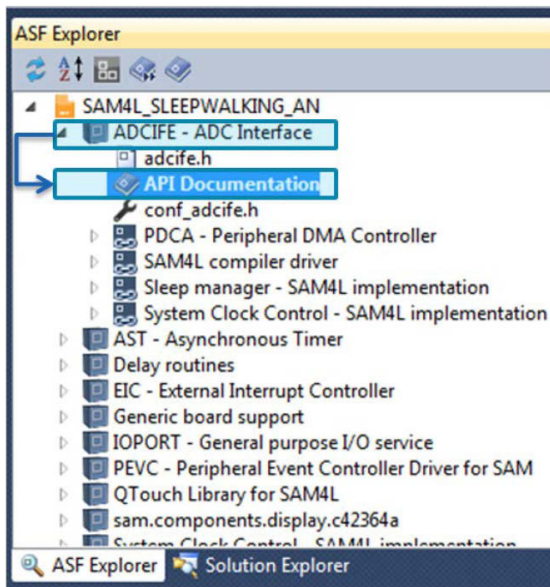
- Click on ASF menu and choose ASF Explorer:



- The ASF Explorer window should appear instead of the Solution explorer. User can switch from Solution to ASF explorers using tabs. The module is then displayed from the ASF Explorer as shown:

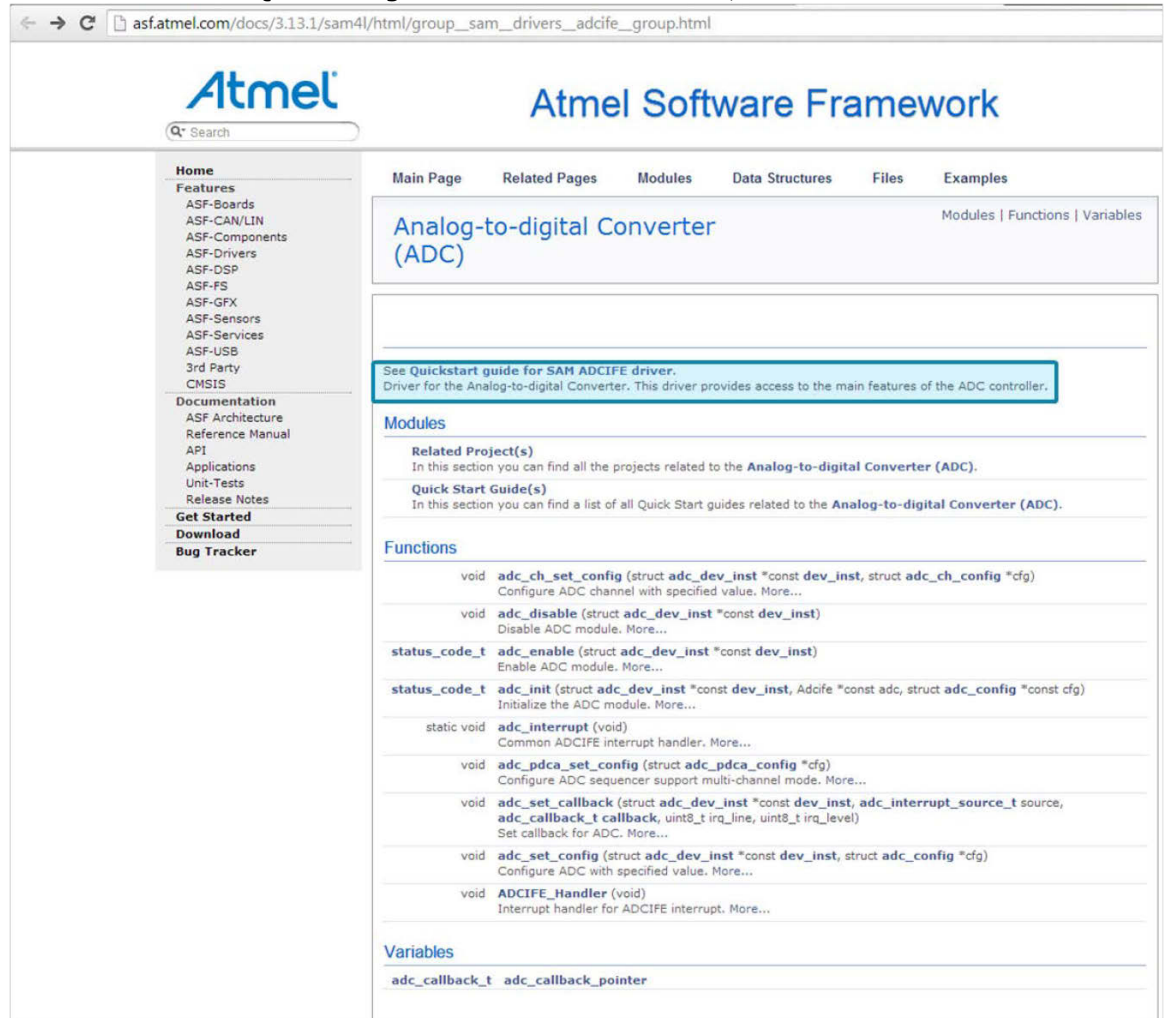


- Deploy the ADCIFE module from this list and click on the API Documentation link, to directly access to the online ADCIFE API documentation:



- The following window should appear from an Internet browser.

- Click on the link: See **Quickstart guide for SAM ADCIFE driver**, as described after



Atmel

Atmel Software Framework

Search

Home

- Features
 - ASF-Boards
 - ASF-CAN/LIN
 - ASF-Components
 - ASF-Drivers
 - ASF-DSP
 - ASF-FS
 - ASF-GFX
 - ASF-Sensors
 - ASF-Services
 - ASF-USB
 - 3rd Party
 - CMSIS
- Documentation
 - ASF Architecture
 - Reference Manual
 - API
 - Applications
 - Unit-Tests
 - Release Notes
- Get Started
- Download
- Bug Tracker

Main Page Related Pages Modules Data Structures Files Examples

Analog-to-digital Converter (ADC)

Modules | Functions | Variables

See Quickstart guide for SAM ADCIFE driver.
Driver for the Analog-to-digital Converter. This driver provides access to the main features of the ADC controller.

Modules

Related Project(s)
In this section you can find all the projects related to the **Analog-to-digital Converter (ADC)**.

Quick Start Guide(s)
In this section you can find a list of all Quick Start guides related to the **Analog-to-digital Converter (ADC)**.

Functions

void	adc_ch_set_config (struct adc_dev_inst *const dev_inst , struct adc_ch_config *cfg)	Configure ADC channel with specified value. More...
void	adc_disable (struct adc_dev_inst *const dev_inst)	Disable ADC module. More...
status_code_t	adc_enable (struct adc_dev_inst *const dev_inst)	Enable ADC module. More...
status_code_t	adc_init (struct adc_dev_inst *const dev_inst , AdcIfc *const adc , struct adc_config *const cfg)	Initialize the ADC module. More...
static void	adc_interrupt (void)	Common ADCIFE interrupt handler. More...
void	adc_pdca_set_config (struct adc_pdca_config *cfg)	Configure ADC sequencer support multi-channel mode. More...
void	adc_set_callback (struct adc_dev_inst *const dev_inst , adc_interrupt_source_t source, adc_callback_t callback, uint8_t irq_line, uint8_t irq_level)	Set callback for ADC. More...
void	adc_set_config (struct adc_dev_inst *const dev_inst , struct adc_config *cfg)	Configure ADC with specified value. More...
void	ADCIFE_Handler (void)	Interrupt handler for ADCIFE interrupt. More...

Variables

adc_callback_t	adc_callback_pointer
-----------------------	-----------------------------

Setup steps

Example code

Add to application C-file:

```
* 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);
* }
```

Workflow

1. Define the interrupt service handler in the application:

```
* 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);
*     }
* }
```

Note

Get ADCIFE status and check if the conversion is finished. If done, read the last ADCIFE result data.

2. Initialize ADC Module:

```
* adc_init(&g_adc_inst, ADCIFE, &adc_cfg);
```

3. Enable ADC Module:

```
* adc_enable(&g_adc_inst);
```

4. Configure ADC single sequencer with specified value:

```
* adc_ch_set_config(&g_adc_inst, &adc_ch_cfg);
```

5. Set callback for ADC:

```
* adc_set_callback(&g_adc_inst, ADC_SEQ_SEOC, adcife_read_conv_result,
*                 ADCIFE_IRQn, 1);
```

Usage steps

Example code

Add to, e.g., main loop in application C-file:

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

Workflow

1. Start ADC conversion on channel:

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

This is the Quickstart guide for the SAM ADCIFE driver, with step-by-step instructions on how to configure and use the driver in a selection of use cases.

The use cases contain several code fragments. The code fragments in the steps for setup can be copied into a custom initialization function, while the steps for usage can be copied into, e.g., the main application function.

Thanks to the API documentation, the process to do is fully based on the workflow described above.

6.3 Code Implementation

6.3.1 Configure the ADCIFE using the Quick Start Guide Workflow

From the API Documentation:

Workflow

1. Define the interrupt service handler in the application:

```
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);
    }
}
```

Note

Get ADCIFE status and check if the conversion is finished. If done, read the last ADCIFE result data.

2. Initialize ADC Module:

```
adc_init(&g_adc_inst, ADCIFE, &adc_cfg);
```

3. Enable ADC Module:

```
adc_enable(&g_adc_inst);
```

4. Configure ADC single sequencer with specified value..

```
adc_ch_set_config(&g_adc_inst, &adc_ch_cfg);
```

5. Set callback for ADC:

```
adc_set_callback(&g_adc_inst, ADC_SEQ_SEOC, adcife_read_conv_result,
                ADCIFE_IRQn, 1);
```

6.3.1.1 Define the ADCIFE Interrupt Handler in the Application

```
/*
 * Include header files for all drivers that have been imported from
 * Atmel Software Framework (ASF).
 */
#include <asf.h>

/* Include sleepwalking_apnote header */
#include "sleepwalking_apnote.h"

/**
 * ADCIFE callback when light sensor detection
 */
void ADCIFE_Callback_sleepwalking(void)
{
    /* clear All interrupt flags for the ADCIFE */
    adc_clear_status(
        &g_adc_inst,
        ADCIFE_SCR_TTO          |
        ADCIFE_SCR_SMTRG        |
        ADCIFE_SCR_WM           |
        ADCIFE_SCR_LOVR         |
        ADCIFE_SCR_SEOC         |
    );

    /* Change the state of the sequential variable to initialize the Active Mode*/
    seq_state = INIT_ACTIVE_MODE;

    /* Switch ON the LED0 to see when interrupt Hits */
    ioport_set_pin_level(PIN_PC10, LED0_ACTIVE_LEVEL);
}
```

6.3.1.2 Configure ADC Module

Before starting the initialization of the ADC module, the ADC instance has to be declared at the beginning of the main file as a global variable:

```
/* Include sleepwalking_appnote header */
#include "sleepwalking_appnote.h"

/* ADC Instance declaration */
struct adc_dev_inst g_adc_inst;
```

In this application, the whole ADCIFE configuration is done by the function `init_ADCIFE()`, **just before** the `app_init`:

- Implement the `init_ADCIFE` function which will be used to perform the workflow Step 2 to 5

```
/**
 * Analog to Digital Converter (ADCIFE) configuration
 */
void init_ADCIFE(void)
{

} //end init ADCIFE

/**
 * Application Initialization function Implementation
 */
static void app_init(void)
{
```

As described from the workflow above several steps are implemented through three dedicated structures which are already declared in the `adcife.h` file:

- The ADC configuration structure (`struct adc_config`)

```
/**
 * Analog to Digital Converter (ADCIFE) configuration
 */
void init_ADCIFE(void)
{
    /** ADC Configuration structure. */
    struct adc_config adc_cfg;
    //0.625*VCC, 75k speed, DIV16, Generic clock sources
    /* System clock division factor is 16 */
    adc_cfg.prescal = ADC_PRESCAL_DIV16;
    /* The Generic clock is used */
    //clkssel = ADC_CLKSEL_GCLK,
    adc_cfg.clkssel = ADC_CLKSEL_GCLK;
    /* Max speed is 75K */
    adc_cfg.speed = ADC_SPEED_75K;
    /* ADC Reference voltage is 0.625*VCC */
    adc_cfg.refsel = ADC_REFSEL_2;
    /* Enables the Startup time */
    adc_cfg.start_up = CONFIG_ADC_STARTUP; //enable the ADC start up value==> the ADC need 12
                                           ADC_CLK at least before starting to convert (see
                                           conf_adcife.h)
```

- The ADC sequencer configuration structure (*struct adc_seq_config*)

```

/** ADC Sequencer Configuration structure. */
struct adc_seq_config adc_seq_cfg;
/* Select Vref for shift cycle */
adc_seq_cfg.zoomrange = ADC_ZOOMRANGE_0;
/* Pad Ground */
adc_seq_cfg.muxneg = ADC_MUXNEG_1;
/* Input PB5 ==> Light Sensor */
adc_seq_cfg.muxpos = ADC_MUXPOS_6;
/* Enables the internal voltage sources */
adc_seq_cfg.internal = ADC_INTERNAL_2;
/* Disables the ADC gain error reduction */
adc_seq_cfg.gcomp = ADC_GCOMP_DIS;
/* Disables the HWLA mode */
adc_seq_cfg.hwla = ADC_HWLA_DIS;
/* 12-bits resolution */
adc_seq_cfg.res = ADC_RES_12_BIT;
/* Enables the single-ended mode */
adc_seq_cfg.bipolar = ADC_BIPOLAR_SINGLEENDED;
/* Use the internal Trig source which would correspond to the peripheral event system and
so the AST */
adc_seq_cfg.trgsel = ADC_TRIG_INTL_TRIG_SRC;
/* Set gain to 0.5 */
adc_seq_cfg.gain = ADC_GAIN_HALF;

```

- The ADC channel configuration structure (*struct adc_ch_config*)

```

/** ADC Channel Configuration structure. */
struct adc_ch_config adc_ch_cfg;
adc_ch_cfg.seq_cfg = &adc_seq_cfg;
/* Internal Timer Max Counter */
adc_ch_cfg.internal_timer_max_count = 60;
/* Window monitor mode is on */
adc_ch_cfg.window_mode = 3; //Window Mode 3: active when LT < result < HT , Channel 6 enabled
adc_ch_cfg.low_threshold = 200;
adc_ch_cfg.high_threshold = 300;

```

- Initialize the previous settings

```

/* Enable The GPIO for the Light Sensor*/
ioport_set_pin_mode(GPIO_PB05A_ADCIFE_AD6, MUX_PB05A_ADCIFE_AD6);
ioport_disable_pin(GPIO_PB05A_ADCIFE_AD6);

/* Apply ADC Configurations */
adc_init(&g_adc_inst, ADCIFE, &adc_cfg);

/* Enable the ADC */
adc_enable(&g_adc_inst);

/* ADC Sequencer Config */
adc_ch_set_config(&g_adc_inst, &adc_ch_cfg);

```



```

    /* Window mode interrupt config */
    adc_set_callback(    &g_adc_inst,
                        ADC_WINDOW_MONITOR,
                        ADCIFE_Callback_sleepwalking,
                        ADCIFE_IRQn, 1);
} //end init ADCIFE

```



INFO

The I/O Port PB05 MUX has to be configured as AD6 input (analog input to be sampled by the ADC); this is done with `ioport_set_pin_mode` and `ioport_disable_pin` above.

- And finally the `app_init` function can be updated as:

```

/**
 * Application Initialization function Implementation
 */
static void app_init(void)
{
    /* Initialize the SAM4L-EK board */
    board_init();

    /* Application's clock setting configuration */
    sysclk_init();

    /* Analog to Digital Converter (ADCIFE) configuration */
    init_ADCIFE();
}

```

7 Capacitive Touch (CATB) Configuration

7.1 Autonomous Touch Overview

The objective of this application is to perform touch measurement without CPU intervention and to wake up the CPU from sleep when the autonomous sensor electrode is touched / untouched. On wakeup, CPU should switch ON/OFF a LED based on the touch status. In this task the SleepWalking feature of the SAM4L device will be used in order to perform Autonomous touch sensing.



INFO

Further details are given from the application note “[Atmel AT04150: QTouch with SAM4L Training Guide](#)”.

As this peripheral does not have its own API documentation, the datasheet has to be used to identify the register to be configured. As a result, the main steps to configure the CATB module are listed below:

- First implement the `init_qtouch` function **just before** the `app_init`
- Enable the CATB Peripheral Clock which is not enabled by default after a RESET (unlike the AST)
- Enable the software reset to reinitialize the CATB module
- Enable the CATB clock at cell level
- Enable Peripheral event in CATB module
- Enable Interrupt at core level
- Enable Interrupt at Peripheral level
- Implement the CATB callback required to wake up the core from wait mode
- Update the `app_init` to call the `init_qtouch` function



INFO

The QTouch library configuration will not be covered in this Application Note. But it is already implemented in the code, QTouch functions are mandatory to set the sensor threshold value, discharge time, etc. The GPIO settings are available in the `touch_sensors_init()` function described in the `touch.c` file located in `/src/qtouch` folder.

7.2 Code Implementation

- First implement the `init_qtouch` function **just before** the `app_init`

```
/**
 * Initialize the CATB and the Qtouch library for the example.
 */
static void init_qtouch(void)
{

} //End init_qtouch

/**
 * Application Initialization function Implementation
 */
static void app_init(void)
{
```

- Set the **CATB** bit in the Power Manager Clock Mask Register System to enable the CATB Peripheral Clock:

```

/**
 * Initialize the CATB and the Qtouch library for the example.
 */
static void init_qtouch(void)
{

    /*Enable CATB clock at peripheral level      */
    sysclk_enable_peripheral_clock(CATB);

```

- Enable the software reset to reinitialize the CATB module by setting the **SWRST** bit field of the CATB configuration register:

```

/* Perform a qtouch Soft reset */
CATB->CATB_CR|=CATB_CR_SWRST;

```

- Set the **EN** bit in CATB configuration Register to enable the CATB Clock at Cell level:

```

/* enable CATB clock at cell level in the control register(EN bit) */
CATB->CATB_CR|=CATB_CR_EN;

```

- Set the **ETRIG** bit in CATB configuration Register, to enable Peripheral event in CATB module:

```

/* Enable Peripheral event in CATB module by setting the ETRIG bit */
CATB->CATB_CR|=CATB_CR_ETRIG;

```

- Initialize the QTouch sensor Library by calling the *touch_autonomous_sensor_enable()* function:

```

/* Configure the QTouch Library to enable the autonomous QTouch detection */
touch_autonomous_sensor_enable();

```

- Adjust the CS0 sensor sensitivity by setting the **CR.ESAMPLE** register bitfield:

```

/* Adjust the Qtouch sensitivity */
CATB->CATB_CR |= CATB_CR_ESAMPLES(13);

```

- Enable **CATB_IRQn** Interrupt at core level:

```

/* Enable CATB IRQn interrupts at core level (NVIC) */
NVIC_ClearPendingIRQ(CATB_IRQn);
NVIC_SetPriority(CATB_IRQn,0);
NVIC_EnableIRQ(CATB_IRQn);

```

- Enable CATB Interrupt at peripheral level by setting the **INTCH** bit for an In Touch detection as described on the next page:

```

/*Enable the in touch (INTCH) IT at Peripheral Level*/
CATB->CATB_IER|=CATB_IER_INTCH;
} //End init_qtouch

```

- At the beginning of the *main.c* file Implement the CATB callback required to wake up the core from wait mode:

```

/**
 * CATB intouch callback.
 */
void CATB_callback_sleepwalking(void)
{
    /* Clear CATB IT */
    CATB->CATB_SCR = CATB_SCR_INTCH;
    while (CATB->CATB_ISR & CATB_ISR_INTCH) {

```

```

    }

    /* Change the state of the sequential variable to initialize the Active
    * Mode*/
    seq_state = INIT_ACTIVE_MODE;
    ioport_set_pin_level(PIN_PC10, LED0_ACTIVE_LEVEL);
} //End CATB_callback_sleepwalking

```

- Update the `app_init` to call the `init_qtouch` function:

```

/**
 * Application Initialization function Implementation
 */
static void app_init(void)
{
    /* Initialize the SAM4L-EK board */
    board_init();

    /* Application's clock setting configuration */
    sysclk_init();

    /* Analog to Digital Converter (ADCIFE) configuration */
    init_ADCIFE();

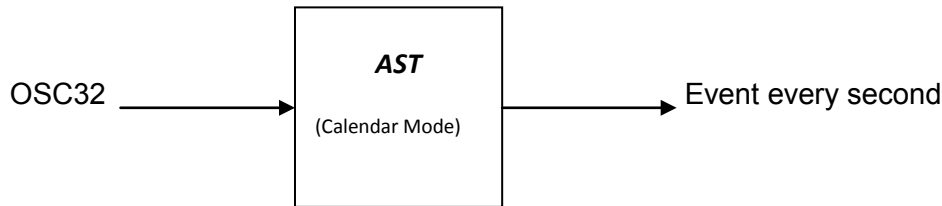
    /* Initialize the CATB module and the Qtouch Library */
    init_qtouch();
}

```

8 Asynchronous Timer (AST) Configuration

In this chapter, we will learn how to configure the Asynchronous Timer of SAM4L device in order to generate a periodic event every second.

To do so, the AST will be configured in Calendar mode and will use the 32kHz as input to generate the event.



It is mandatory to use an external 32kHz oscillator in order to have the best precision for our watch application.

As for most of the integrated peripherals, the AST initialization should respect a specific workflow.

- First implement the `init_ast` function **just before** the `app_init`
- Enable first the OSC32K (AST clock source without prescaler)
- Enable the AST clock at cell level
- Configure the AST registers to set the AST as
 - Set the AST in counter mode
 - Set the OSC32K as the AST clock source
 - Initialize the AST counter value to 0
- Apply AST configuration
- Initialize Calendar value
- Set Periodic event 0 every seconds
- Enable the Event Interface for Peripheral Event system use in the AST Level
- Update the `app_init` to call the `ast_qtouch` function



INFO

The AST Peripheral Clock is already enabled after a RESET.

8.1 Code Implementation

- First implement the `init_ast` function **just before** the `app_init`.

```
/**
 * Initialize the Asynchronous Timer (AST) in calendar mode using
 * the 32Khz oscillator as clock source.
 */
void init_ast(void)
{

} //end init_ast

/**
 * Application Initialization function Implementation
 */
static void app_init(void)
{
```

- Enable first the OSC32K (AST clock source without prescaler)

```

/**
 * Initialize the Asynchronous Timer (AST) in calendar mode using
 * the 32KHz oscillator as clock source.
 */
void init_ast(void)
{

    osc_enable(OSC_ID_OSC32);
    /* wait while the OSC has not started */
    while(!osc_is_ready(OSC_ID_OSC32));
}

```

- Enable the AST clock at cell level

```

/*Enable the AST clock at peripheral bridge level and at cell level*/
ast_enable(AST);

```

- Configure the AST registers to set the AST as:
 - Set the AST in calendar mode
 - Set the OSC32K as the AST clock source
 - Initialize the AST calendar value to “calendar” (see annexes sleepwalking_appnote.h) file for the calendar definition

```

/*Configure the AST registers to
    • Set the AST in counter mode
    • Set the OSC32KHz as the AST clock source
    • Initialize the AST counter value to 0 */

struct ast_config ast_conf;
ast_conf.mode = AST_CALENDAR_MODE; // we set the AST in CALENDAR mode
ast_conf.osc_type = AST_OSC_32KHZ; //The OSC32KHz is the AST clock source
ast_conf.psel = AST_PSEL_32KHZ_1HZ; //We want a 1s clock frequency
ast_conf.calendar = calendar; // the calendar struct is used to set up the calendar
                                register

```

```
void ast_set_config(&ast_cfg);
```

Where ast_cfg is an instance of the structure “ast_config”, defined below:

- | | |
|----------------------------------|---|
| • ast_mode_t mode | Set counting mode |
| • ast_oscillator_type_t osc_type | Set oscillator input |
| • uint8_t psel | Set counting Prescaler |
| • uint32_t counter | Set initial counter value(counter mode) |
| • struct ast_calendar calendar | Set initial calendar value(calendar mode) |

- Apply AST configuration

```

/* Apply AST configuration */
ast_set_config(AST, &ast_conf);

```

- Initialize Calendar value

```

/* Initialize Calendar value*/
ast_write_calendar_value(AST, calendar);

```

```
// Set Periodic event
void ast_write_periodic0_value (Ast *ast, uint32_t pir)
```

The pir argument allows setting periodic event frequency according to following formula:

$$f_{periodic_event} = \frac{f_{osc}}{(2^{pir+1})}$$



INFO

Set Periodic event 0 every seconds.

```
/*Set Periodic event 0 every 0.5 second INSEL bitfield of the Periodic interval register */
ast_write_periodic0_value(AST, AST_PSEL_32KHZ_1HZ-1);
```



INFO

Enable the Event Interface for Peripheral Event system use in the AST Level.

```
/*Enable the Event Interface for Peripheral Event system use in the AST Level*/
ast_enable_event(AST, AST_EVENT_PER);
} //end init_ast
```



INFO

Update the `app_init` to call the `init_qtouch` function.

```
/**
 * Application Initialization function Implementation
 */
static void app_init(void)
{
    /* Initialize the SAM4L-EK board */
    board_init();

    /* Application's clock setting configuration */
    sysclk_init();

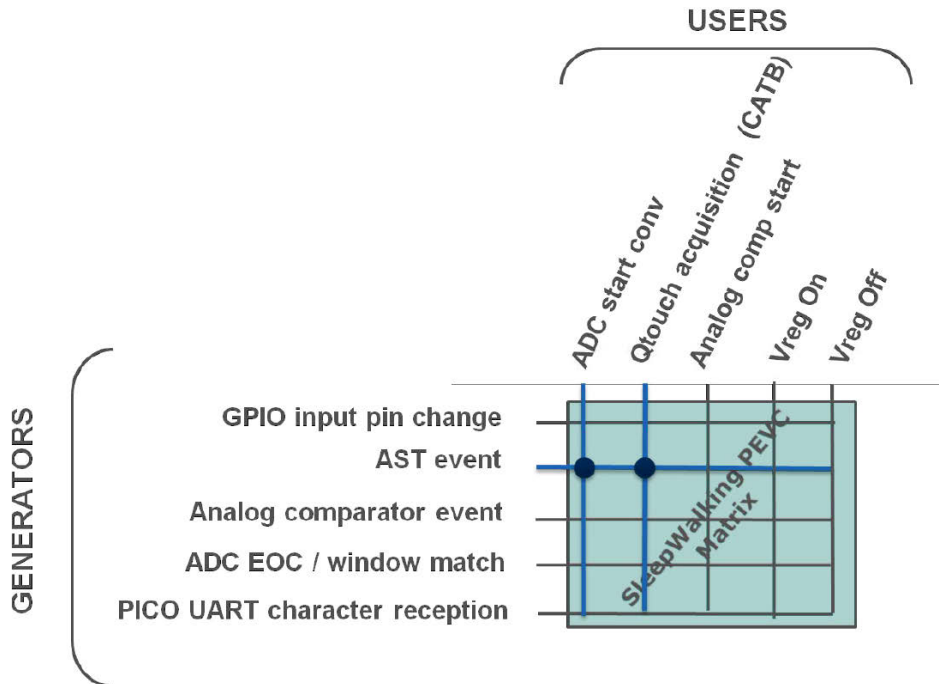
    /* Analog to Digital Converter (ADCIFE) configuration */
    init_ADCIFE();

    /* Initialize the CATB module and the Qtouch Library */
    init_qtouch();

    /* Initialize & configure the AST */
    init_ast();
```


9 Peripheral Event System Controller (PEVC) Configuration

The aim of this Peripheral is to interconnect the AST trigger event to the CATB autonomous QTouch sensing and to the ADCIFE Start Conversion triggering as described below:



The main steps to do this are listed below:

- First implement the `init_PEVC` function **just before** the `app_init`
- Declare a peripheral event channel configuration structure
- Enable the PEVC Peripheral Clock which is not enabled by default after a RESET (unlike the AST)



INFO

From this point, one can be noticed that the same steps are reproduced for the ADCIFE.

- Get the event channel default config to initialize the structure
- Configure the PEVC channel to link the
 - AST - Periodic Event 0 (Generator Channel no. 8) to the CATB - Trigger one autonomous touch sensing Event (User Channel no. 6), refer to the datasheet for more details
- Enable the PEVC channel which corresponds to CATB trigger one autonomous touch sensing
- Update the `app_init` to call the `init_PEVC` function

9.1 Code Implementation

- First implement the `init_PEVC` function **just before** the `app_init`:

```
/**
 * Initialize the Peripheral Event Controller for the CATB.
 */
void init_PEVC(void)
{

} //end init_PEVC
```

```

/**
 * Application Initialization function Implementation
 */
static void app_init(void)
{

```

- Declare a peripheral CATB event channel configuration structure

```

/**
 * Initialize the Peripheral Event Controller for the CATB and the ADCIFE.
 */
void init_PEVC(void)
{
    /*declare the peripheral event channel configuration structure */
    struct events_ch_conf ch_config;

```

- Declare a peripheral ADCIFE event channel configuration structure

```

/*declare the peripheral ADCIFE event channel configuration structure */
struct events_ch_conf ch_config_adcife;

```

- Enable the PEVC Peripheral Clock which is not enabled by default after a RESET (unlike the AST)

```

/*Enable the PEVC Peripheral Clock which is not
enabled by default after a RESET (contrary to the AST) */
events_enable();

```



INFO

From this point, one can be noticed that the same steps are reproduced for the ADCIFE.

- Get the event channel default config to initialize the structure

```

/*Configure the PEVC channel to link the AST Periodic
event 0 to the Peripheral Event system CATB trigger one autonomous
touch sensing */
events_ch_get_config_defaults(&ch_config_catb);

```

- Configure the PEVC channel to link the
 - AST - Periodic Event 0 (Generator Channel no. 8) to the CATB - Trigger one autonomous touch sensing Event (User Channel no. 6), refer to the datasheet for more details

```

ch_config_catb.channel_id = PEVC_ID_USER_CATB;
ch_config_catb.generator_id = PEVC_ID_GEN_AST_2;
ch_config_catb.shaper_enable = true;

/*      call the events_ch_configure() function, defined
in events.h to configure the PEVC channel new parameters*/
events_ch_configure(&ch_config_catb);

```

- Enable the PEVC channel which corresponds to CATB trigger one autonomous touch sensing

```

/*Enable the EVENT channel dedicated to the CATB*/
events_ch_enable(PEVC_ID_USER_CATB);

```

- Reproduce the three previous steps for the ADCIFE
 - Get the event channel default config to initialize the structure

```

/*Configure the PEVC channel to link the AST Periodic
event 0 to the Peripheral Event system ADCIFE trigger one autonomous
touch sensing */
events_ch_get_config_defaults(&ch_config_adcife);

```

- Configure the PEVC channel to link the
 - AST - Periodic Event 0 (Generator Channel no. 8) to the ADCIFE - Trigger one autonomous touch sensing Event (User Channel no. 4), refer to the datasheet for more details

```

ch_config_adcife.channel_id = PEVC_ID_USER_ADCIFE_SOC;
ch_config_adcife.generator_id = PEVC_ID_GEN_AST_2;
ch_config_adcife.shaper_enable = true;

/*      call the events_ch_configure() function, defined
in events.h to configure the PEVC channel new parameters*/
events_ch_configure(&ch_config_adcife);

```

- Enable the PEVC channel which corresponds to CATB trigger one autonomous touch sensing

```

/*Enable the EVENT channel dedicated to the ADCIFE*/
events_ch_enable(PEVC_ID_USER_ADCIFE_SOC);
} //end init_PEVC

```

- Update the `app_init` to call the `init_PEVC` function (`ast_init` must remain as the last init function called because the counting is supposed to start just after...)

```

/**
 * Application Initialization function Implementation
 */
static void app_init(void)
{
    /* Initialize the SAM4L-EK board */
    board_init();

    /* Application's clock setting configuration */
    sysclk_init();

    /* Analog to Digital Converter (ADCIFE) configuration */
    init_ADCIFE();

    /* Initialize the CATB module and the Qtouch Library */
    init_qtouch();

    /* Initialize the peripheral Event System Controller allowing
     * Peripherals to request theirs clocks */
    init_PEVC();

    /* Initialize & configure the AST */
    init_ast();

```

10 External Interrupt Controller (EIC) Configuration

In this paragraph, the External Interrupt Controller will be configured in order to use the PB0 push button to generate an External Interrupt allowing the core to go back into WAIT mode.

To correctly configure the External Interrupt Controller (EIC), the main steps to do this are listed below:

- First implement the `init_eic` function **just before** the `app_init`
- Declare an EIC config structure variable
- Configure the structure
- Enable the peripheral clock of the EIC module
- Apply the configuration
 - Interrupt type configuration
 - Edge triggered
 - Falling edge
 - low level
 - Filter disabled
 - Asynchronous Mode
- Callback implementation:
- Enable the corresponding line
- Update the `app_init` to call the `init_eic` function
- Finish the `app_init` implementation by calling the `black_box_app_init` function

10.1 Code Implementation

- First implement the `init_eic` function **just before** the `app_init`

```
/**
 * Initialize the External Interrupt Controller for the example.
 */
void init_eic(void)
{

} //end init_eic

/**
 * Application Initialization function Implementation
 */
static void app_init(void)
{
```

- Declare an EIC config structure variable

```
void init_eic(void)
{

    /*Configure EIC module to as described previously*/
    struct eic_line_config eic_line_conf;
```

- Configure the structure
 - Interrupt type configuration
 - Edge triggered
 - Falling edge

- low level
- Filter disabled
- Asynchronous Mode

```

/* set the EIC mode Triggered*/
eic_line_conf.eic_mode = EIC_MODE_EDGE_TRIGGERED;

/* set the EIC mode detection on falling edge*/
eic_line_conf.eic_edge = EIC_EDGE_FALLING_EDGE;

/* EIC_LEVEL_LOW_LEVEL is defined in eic.h to set the EIC mode detection on low level */
eic_line_conf.eic_level = EIC_LEVEL_LOW_LEVEL;

/*disable the EIC input filter */
eic_line_conf.eic_filter = EIC_FILTER_DISABLED;

/* enable the EIC asynchronous mode detection */
eic_line_conf.eic_async = EIC_ASYNC_MODE;

```

- Enable the peripheral clock of the EIC module

```

/*Enable the peripheral clock of the EIC module */
eic_enable(EIC);

```

- Apply the configuration

```

/*Apply the interrupt configuration*/
eic_line_set_config(EIC, GPIO_PUSH_BUTTON_EIC_LINE, &eic_line_conf);

```

- Callback implementation

```

/*Enable the peripheral clock of the EIC module */
eic_line_set_callback(EIC, GPIO_PUSH_BUTTON_EIC_LINE, Button_Callback, EIC_5_IRQn, 1);

```

- Enable the corresponding line

```

/*Enable the corresponding line */
eic_line_enable(EIC, GPIO_PUSH_BUTTON_EIC_LINE);
} //end of init EIC

```

- Update the `app_init` to call the `init_eic` function

```

/**
 * Application Initialization function Implementation
 */
static void app_init(void)
{
    /* Initialize the SAM4L-EK board */
    board_init();

    /* Application's clock setting configuration */
    sysclk_init();

    /* Analog to Digital Converter (ADCIFE) configuration */
    init_ADCIFE();

    /* Initialize the CATB module and the Qtouch Library */

```

```

init_qtouch();

/* Initialize the peripheral Event System Controller allowing
 * Peripherals to request theirs clocks */
init_PEVC();

/* Initialize the EIC module */
init_eic();

```

- Finish the `app_init` implementation by calling the `black_box_app_init` function

```

/* Initialize the EIC module */
init_eic();

/* Initialize the black box application */
black_box_app_init();

/* Initialize & configure the AST */
init_ast();
} //end app_init

```

11 Test the App Init Function Implementation

- **Build**, program and run the application

— In order to build the project, **click** on the Build button:



Make sure the SAM4L-EK board is connected to your PC with a micro-USB cable through the J1 connector.

- Then download the program in the internal flash of the SAM4L by **clicking** on the *Start Debugging and break* button:

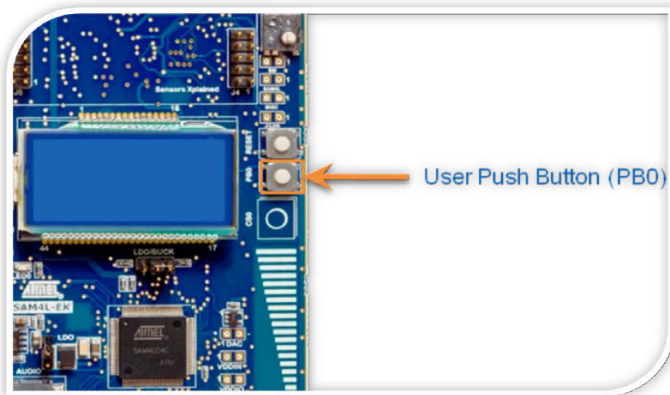


Atmel Studio will ask you to select the Debug Tool.

- **Select** the on-board *J-Link* (note that the serial number in parentheses differs from one board to another):
- Once programmed, **start** the code execution by **clicking** on the green arrow:



Set the clock of your application by using PB0.



- Press PB0 to set the Minutes



- Hold PB0 for a short duration (~2 seconds) to set Hours
- Press PB0 to set the Hours



- Hold PB0 for a short duration (~2 seconds) to start the clock
- Touching the CS0 QTouch button should lit the LED0

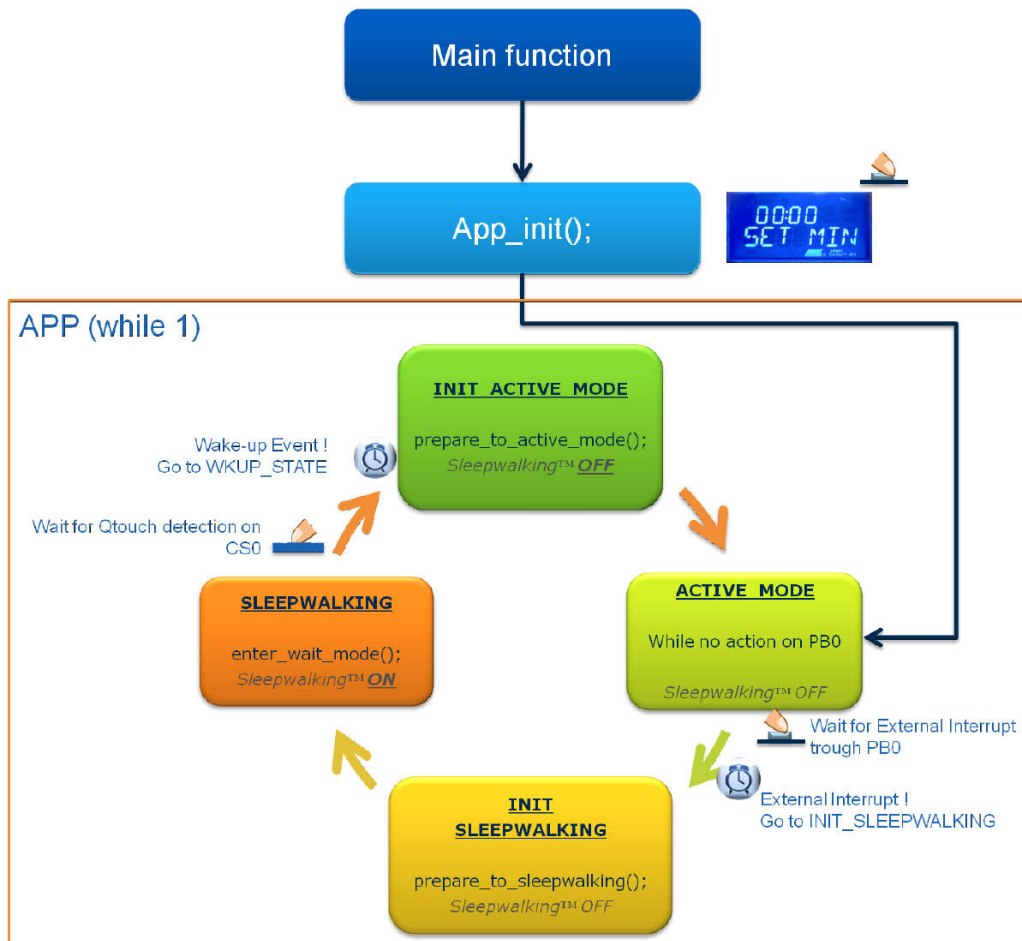
RESULT The application is running correctly and uses the PLL + OSC0 oscillator to run @48MHz.



When the debug session is running, the *Stop* button  stops program execution and exits the debug session. If you want to stop the program but keep the debug session active, simply click on the *Pause* button .

12 State Machine Implementation

The last step of the code implementation is the state machine as described in Figure 2-5. At this step the App init function has been implemented successfully. The state machine requires other functions to be implemented as described in the updated representation below:



The **app** function will be executed in the infinite loop. It is only composed of a state machine based on a switch/case algorithm. Each case is defined by a particular state of the state machine, and the state is updated after each function called from a state to another, till reaching the SLEEPWALKING state, which requires an interrupt either coming from the ADCIFE or from the CATB, waking up the core and exit from the WAIT mode.

The first step will be to implement the app function as described in the figure above.

After this step, three functions remain to be implemented allowing the SleepWalking application to be functional:

- `prepare_to_active_mode`
- `prepare_to_sleepwalking`
- `enter_wait_mode`

12.1 Code Implementation

12.1.1 Implement the App Function

- Just before the main function, implement the app function which contains the state machine

```
/**
 * State Machine Function
```

```

* Sleepwalking Application State machine
*/
void app(void)
{
    switch (seq_state) {
        case INIT_SLEEPWALKING:
            /* Prepare the application to enter in Sleepwalking */
            prepare_to_sleepwalking();
            break;

        case SLEEPWALKING:
            /* Sleepwalking : enter in wait mode... */
            enter_wait_mode();
            break;

        case INIT_ACTIVE_MODE:
            /* Prepare the application to enter in active mode */
            prepare_to_active_mode();
            break;

        case ACTIVE_MODE:
            while (seq_state == ACTIVE_MODE) {
                /* use PB0 to hit interrupt to perform the
                 *break */
            }
            break;

        default:
            break;
    }
} //end App

/*****
/*                               MAIN Function                               */
*****/
void main(void)

```

- Declare the `seq_state` variable as global volatile at the beginning of the `main.c` file:

```

/*****
/*                               Global var                               */
*****/
/*      Initial State      */
volatile char seq_state = INIT_ACTIVE_MODE;

/* ADC Instance declaration */
struct adc_dev_inst g_adc_inst;

```

12.1.2 ACTIVE_MODE Implementation

This state is the RUN mode of the application where:

- The main clock is 48MHz
- The LCD display is ON with the clock time displayed on it
- We stay in this mode while an external interrupt event (EIC controller) is not detected (PB0 pushed)

12.1.3 Prepare to SleepWalking Implementation

This state allows preparing the SAM4L device to go back to Wait mode, and configure peripherals to perform SleepWalking by:

- Implement the `prepare_to_sleepwalking` function just before the `app` function
- Disable PDCA used to manage transfer data to the LCD while the LCD is OFF to avoid extra power consumption in SleepWalking
- Disable the LCD clock and the LCD Back Light to avoid extra power consumption in SleepWalking
- Restore the slow clock running @12MHz with FastRC as clock source (PS1 available)
- Enable the CATB module clock to allow QTouch interrupt
- Configure the ADC
- Initialize the AST
- Enable the Peripheral Event Controller to interconnect AST, CATB, and the ADC
- Disable the external the External Interrupt Controller (EIC) to avoid interrupt coming from PB0
- Change the State machine state to go into SLEEPWALKING

12.1.3.1 Code Implementation

- Implement the `prepare_to_sleepwalking` function just before the `app` function:

```
/**
 * Prepare the application to enter in Sleepwalking
 */
static void prepare_to_sleepwalking(void)
{

} //end prepare_to_sleepwalking

/**
 * State Machine Function
 * Sleepwalking Application
 */
void app(void)
{
```

- Prepare the “Black Box” application for SleepWalking reducing power consumption:
 - Disable PDCA used to manage transfer data to the LCD while the LCD is OFF to avoid extra power consumption in SleepWalking
 - Disable the LCD clock and the LCD Back Light to avoid extra power consumption in SleepWalking

```
/**
 * Prepare the application to enter in Sleepwalking
 */
static void prepare_to_sleepwalking(void)
{

    /* Prepare the Black Box application to reduce consumption*/
    black_box_prepare_to_sleepwalking();
```

- Restore the slow clock running @12MHz with FastRC as clock source to make power scaling mode PS1 available

```
/* Change the clock source by Switching to the FAST RC clock running at 12MHz */
switch_to_slower_clock();
```



INFO

See the sleepwalking_appnote.c file to see the switch to slower clock function implementation.

- Enable the CATB module clock to allow QTouch interrupt

```
/* Enable CATB clock at peripheral level */
sysclk_enable_peripheral_clock(CATB);
```

- Enable the ADCIFE module clock

```
/* Enable ADCIFE if used */
adc_enable(&g_adc_inst);
```

- Initialize the AST

```
/* Reinitialize the AST & the PEVC */
init_ast();
```

- Enable the Peripheral Event Controller to interconnect AST, CATB and the ADC

```
/* Reinitialize the AST & the PEVC */
init_ast();
init_PEVC();
```

- Disable the external the External Interrupt Controller (EIC) to avoid interrupt coming from PB0 during SleepWalking state

```
/* Disable EIC interrupt to avoid external interrupt from PB0 */
eic_enable(EIC);
eic_line_disable(EIC, GPIO_PUSH_BUTTON_EIC_LINE);
eic_disable(EIC);
```

- Change the State machine state to go into SLEEPWALKING

```
/* Go to Wait mode */
seq_state = SLEEPWALKING;
} //end prepare_to_sleepwalking
```

12.1.4 SLEEPWALKING State Implementation

To perform SleepWalking, the SAM4L has to embed flexible capabilities regarding its peripheral clock management and must be able to get a higher modularity in power consumption versus performance ratio. This is done with its embedded Low power techniques: Power Saving and Power Scaling.

In this mode, the CPU clock is OFF, the core is in WAIT Mode, and only the 32kHz oscillator is enabled for the AST trig event activity.

12.1.4.1 Code Implementation

- Implement the `enter_wait_mode` just before the `app` function

```
**
* Function used to enter in wait mode used in Sleepwalking state
* with power scaling 1 and Fast wake up enabled
```

```

*/
void enter_wait_mode(void)
{

} //end enter_wait_mode

/**
 * State Machine Function
 * Sleepwalking Application State machine
 */
void app(void)
{

```

- Change Power Scaling mode to be in very low power mode

```

**
 * Function used to enter in wait mode used in Sleepwalking state
 * with power scaling 1 and Fast wake up enabled
 */
void enter_wait_mode(void)
{

    /*Change Power scaling to be in very low power mode */
    bpm_power_scaling_cpu(BPM, BPM_PS_1);

```

- Power Saving is changed (WAIT mode) if the PMCON register is unlocked

```

/* Unlock the BPM PMCON register before modifying it */
BPM_UNLOCK(PMCON);

```

- Enable the fast wake-up capability by setting the FASTWKUP bit in the PMCON register

```

/*enable the fast wake up capability by setting the FASTWKUP bit in the PMCON register*/
bpm_enable_fast_wakeup(BPM);

```

- Enter in WAIT mode

```

/* Enter in wait mode */
bpm_sleep(BPM, BPM_SM_WAIT);
} //end enter_wait_mode

```

12.1.5 INIT_ACTIVE_MODE State Implementation

Once in the state machine, this state is an intermediary state which is used just after wake up (touch on CS0) and allows configuring the peripheral used in ACTIVE state such as:

- Switch the clock to full speed to have a powerful data processing
- Disable the CATB module clock
- Enable PDCA used to manage transfer data to the LCD
- Enable and Initializes the LCD controller
- Enable the External Interrupt Controller (EIC)
- Enable LCD Back Light to display the time
- Change the State machine state to go into ACTIVE_MODE

12.1.5.1 Code Implementation

- Implement the `prepare_to_active_mode` just before the `app` function

```
/**
 * Prepare the application to enter in Active mode
 */
void prepare_to_active_mode(void)
{

} //end prepare_to_active_mode

/**
 * State Machine Function
 * Sleepwalking Application State machine
 */
void app(void)
{
```

- Switch the clock to full speed to have a powerful data processing (no power constraints in active mode)

```
/**
 * Prepare the application to enter in Active mode
 */
void prepare_to_active_mode(void)
{
    /* switch to Full speed mode (PLL+OSC0) making available powerful data processing capability
    */
    switch_to_full_speed_clock();
```

- Disable the CATB module clock

```
/* Disable CATB clock at peripheral level to avoid qtouch interrupt*/
sysclk_disable_peripheral_clock(CATB);
```

- Disable the ADCIFE module clock

```
/* Disable ADCIFE clock at peripheral level to avoid Light sensor interrupt if ADCIFE is
used*/
adc_disable(&q_adc_inst);
```

- Prepare the black box application to the active mode
 - Enable PDCA used to manage transfer data to the LCD
 - Enable and Initializes the LCD controller
 - Enable LCD Back Light to display the time

```
/* Prepare the Black Box application to active mode */
black_box_prepare_to_active();
```

- Enable the External Interrupt Controller (EIC)

```
/* Init the External interrupt Controller */
init_eic();
```

- Change the State machine state to go into ACTIVE_MODE

```
/* Go in Run Mode */
seq_state = ACTIVE_MODE;
} //end prepare_to_active_mode
```


12.1.6 Update the Main Function to Call the App Function

```
//end App

/*****
/*                                */
/*****
void main(void)
{
    /* Initialize the System and the App */
    app_init();

    while (1) {
        app();
    }
}
```

13 Conclusion

In this application note you have discovered the SAM4L main features used to perform Sleep Walking, such as:

- Low power techniques: Power Saving and Power Scaling
- Peripheral Clock Management flexibility
- Peripheral Event System

These main features make the SAM4L more flexible regarding its peripheral clock management and able to get a higher modularity in power consumption versus performance ratio.

You are now familiar with how to use the AST to generate a trigger event clock source. You configured the Peripheral Event System controller (PEVC) to link this trigger source to a user such as CATB or ADCIFE. And finally you enabled the interrupt to wake up the core or to go back to sleep.

All these features are mandatory to perform the SAM4L SleepWalking mode.

14 Suggested Reading

- [QTouch with SAM4L training Guide](#)
- [Getting Started with SAM4L](#)
- [SAM4L-EK User guide](#)
- [SAM4L-EK Production Files](#)

14.1 Device Datasheet

The device datasheet contains block diagrams of the peripherals and details about implementing firmware for the device. It also contains the electrical specifications and expected characteristics of the device.

The datasheet is available on <http://www.atmel.com/> in the Datasheets section of the product page.

14.2 Evaluation Kit User Guide

The SAM4L-EK user guide contains schematics that can be used as a starting point when designing with the SAM4L product family devices. This user guide is available on <http://www.atmel.com/> in the documents section of the SAM4L-EK page.

14.3 ARM Documentation on Cortex-M4 Core

- [Cortex-M4 Devices Generic User Guide](#)
- [Cortex-M4 Technical Reference Manual](#)
- These documents are available at <http://www.arm.com/> in the info center section

15 Revision History

Doc Rev.	Date	Comments
42320A	5/2014	Initial document release.



Atmel Corporation 1600 Technology Drive, San Jose, CA 95110 USA T: (+1)(408) 441.0311 F: (+1)(408) 436.4200 | www.atmel.com

© 2014 Atmel Corporation. /
Rev.:Atmel-42320A-Implementing-SleepWalking-on-ARM-Cortex-M4-MCU-Application-Step-by-step-Project-Building-Guide-ApplicationNote_052014.

Atmel®, Atmel logo and combinations thereof, AVR®, Enabling Unlimited Possibilities®, megaAVR®, picoPower®, QTouch®, XMEGA®, and others are registered trademarks or trademarks of Atmel Corporation in U.S. and other countries. ARM®, ARM Connected® logo, Cortex®, and others are the registered trademarks or trademarks of ARM Ltd. Other terms and product names may be trademarks of others.

DISCLAIMER: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN THE ATMEL TERMS AND CONDITIONS OF SALES LOCATED ON THE ATMEL WEBSITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS AND PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and products descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

SAFETY-CRITICAL, MILITARY, AND AUTOMOTIVE APPLICATIONS DISCLAIMER: Atmel products are not designed for and will not be used in connection with any applications where the failure of such products would reasonably be expected to result in significant personal injury or death ("Safety-Critical Applications") without an Atmel officer's specific written consent. Safety-Critical Applications include, without limitation, life support devices and systems, equipment or systems for the operation of nuclear facilities and weapons systems. Atmel products are not designed nor intended for use in military or aerospace applications or environments unless specifically designated by Atmel as military-grade. Atmel products are not designed nor intended for use in automotive applications unless specifically designated by Atmel as automotive-grade.