Atmel AVR2059: BitCloud Porting Guide

8-bit Atmel Microcontrollers

Application Note

Features

- **Instructions on porting BitCloud to a custom board**
- **Description of BitCloud build process**
- **Porting a reference application to another development board and to a custom board**
- **Instructions for creating Board Support Package from scratch**

Introduction

This document explains how to run BitCloud on a custom board which is different from evaluation and reference boards not included in supported kits. Such instructions may be particularly useful when the application development moves from prototyping to field trials and real-world deployments or whenever the application is moved to real-world devices considerably different from evaluation kits.

Assuming that the BitCloud SDK supports the MCU and the radio chip combination used on a custom board, the stack and the application can be hosted on such a board with little difficulty. However, moving to a non-standard board may require minor re-configuration and/or modification of stack components that deal with hardware, especially the Hardware Abstraction Layer and the Board Support Package. The following chapters outline common modifications that may be required as well as cover related topics such as application build process.

Rev. [doc nr]-AVR-05/11

1 Basic Principles

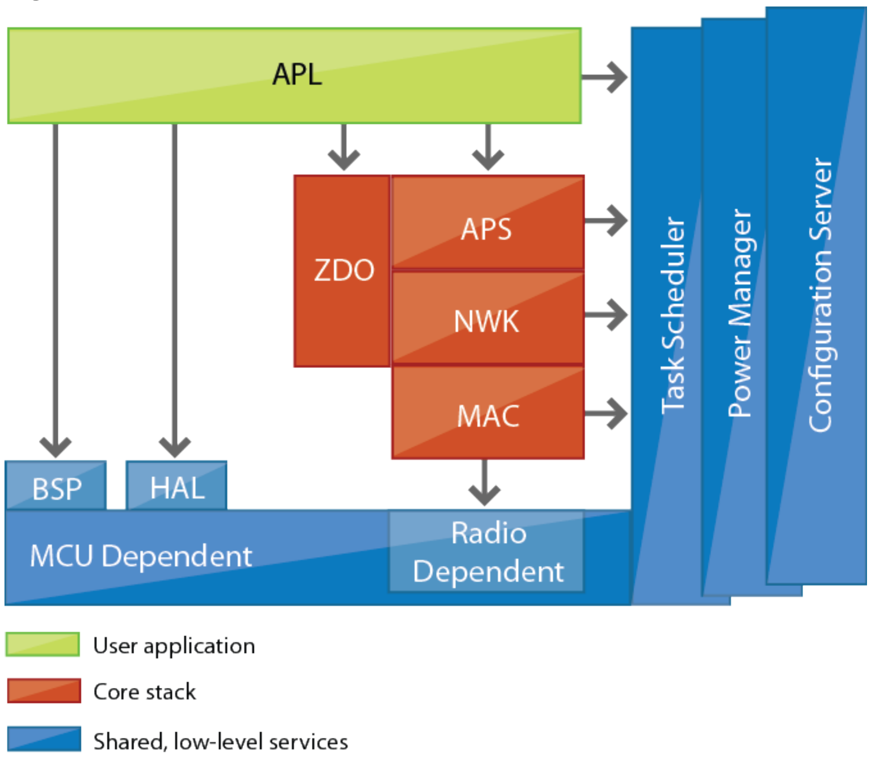
1.1 Architecture

The BitCloud stack is Atmel's full-feature implementation of ZigBee PRO protocol. BitCloud supports a number of Atmel development and evaluation boards; for complete list refer to [1] or [2]. Applications images provided with the SDKs or compiled from sample application sources can run on these boards without any additional changes. User's custom applications can also be developed, compiled, and used with the boards without any changes to BitCloud configuration.

Still, most user applications are eventually ported and run on custom boards different from the standard development and evaluation boards provided with the kits. Since custom boards often feature different interfaces, pins and peripherals configuration, the standard BitCloud configuration needs to be adjusted to fit the hardware platform.

In general this is possible as long as BitCloud libraries support the MCU and radio chip combination used on the custom board.

Figure 1-1 BitCloud Stack Architecture



BitCloud architecture is shown on Figure 1-1. Two main components interacting directly with the hardware are **Hardware Abstraction Layer (HAL)** and **Board Support Package (BSP)**. The hardware-dependent part of the MAC layer also contains some board specific configuration, but these configurations are exposed in HAL and can be modified without recompiling the BitCloud library.

In the above architecture, HAL and BSP serve slightly different purposes. While HAL manages MCU's interfaces enabling interaction over particular pins, BSP is used to control board peripherals. Both HAL and BSP are provided in source code and can be easily modified by the user. Source code is also provided for Persistent Data Server (PDS), Configuration Server (CS), as well as drivers.

1.2 Application Build Process

To simplify the building process all sample applications provided with BitCloud SDK are equipped with necessary configuration and project files for two supported toolchains:

- AVR Studio and the GCC compiler. Applications are compiled by invoking the make utility which relies on the application makefile.
- IAR Embedded Workbench and supporting compiling tools. Within this toolchain two approaches are possible. The user can compile applications:
    - from the command line by running the make utility which relies on the application makefile
    - directly from IAR Embedded Workbench (by setting compile options with IAR Workbench integrated configuration tools)

The following two sections explain in detail how different compilation methods are structured. The toolchains supported by each platform are listed in [1] and [2].

Whatever the toolchain used, all reference applications' additional configuration parameters are collected in the `configuration.h` file located in the root application directory. This file is commonly used to set values for Configuration Server parameters and is referred to throughout this document.

The standard application build process automatically compiles BSP, PDS, ConfigServer, and drivers. The HAL component shall be compiled separately as described in Section 2.1.2.

1.2.1 Building applications using makefiles

Each sample application is provided with makefiles for the most typical application configurations. Makefiles are located in the `\makefiles` directory inside subdirectories corresponding to different supported boards. In addition to these low-level makefiles each application includes high-level Makefile located in the application root folder.

The high-level Makefile is used to specify the low-level makefile that will be used to build the application. The choice depends on the values assigned to special variables inside the high-level Makefile:

- `PROJECT_NAME`: specifies the subdirectory name of the `\makefiles` directory where the target file is located
- `CONFIG_NAME`: used to obtain the target makefile name by adding `CONFIG_NAME` to `Makefile_`.

For example, if Makefile contains the following lines:

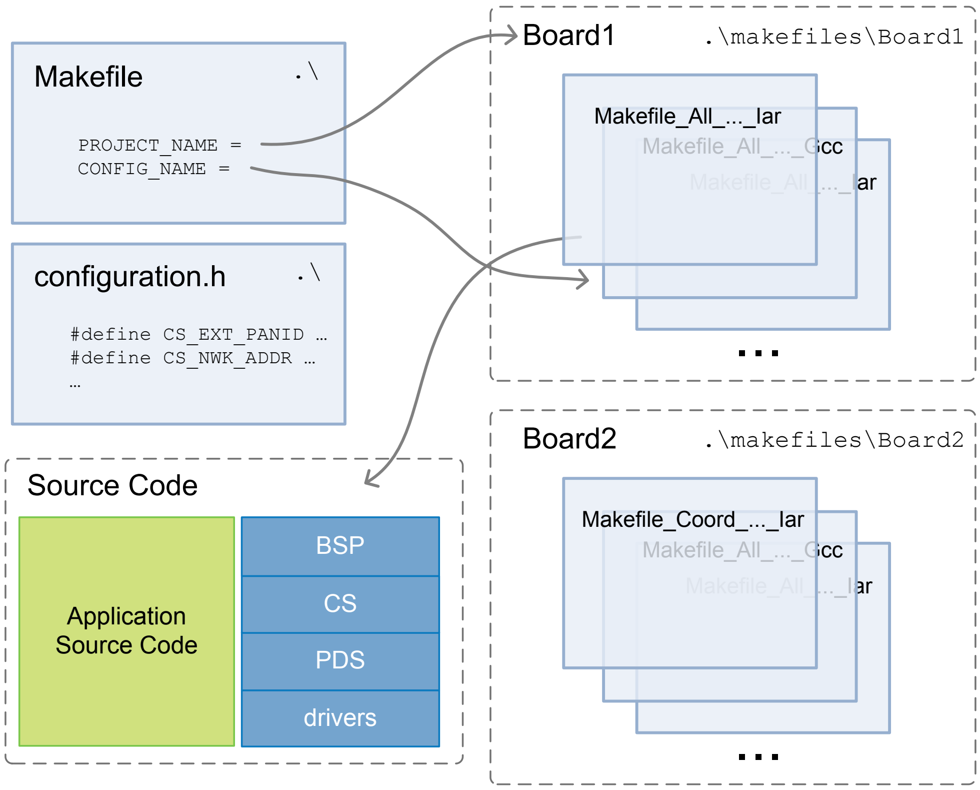
```
PROJECT_NAME = MeshBean
CONFIG_NAME = All_ZigBit_Atmega1281_Rf230_8Mhz_Gcc
```

then the compilation instructions will be extracted from the makefile located at

```
\makefiles\MeshBean\Makefile_All_ZigBit_Atmega1281_Rf230_8Mhz_Gcc
```

Application structure is illustrated in Figure 1-2. A high-level Makefile for sample applications already contains commented lines for all configurations provided, so the user just has to uncomment appropriate lines.

Figure 1-2 Application Build Structure with Makefiles



After desired configuration is chosen in the Makefile, the application can be built by executing `make clean all` from the command line in the application root folder or by selecting the `Build` command in the context menu in AVR Studio.

1.2.1.1 Low-level makefile name structure

The name of a low-level makefile consists of parts showing which configuration the file specifies. These include specification of

- Device type (`All`, `Coordinator`, `Router`, or `EndDevice`); `All` means that this configuration can be used for any device type
- Module or board (`ZigBit`, `STK600`, etc.)
- MCU (`Atmega1281`, `Atmega128rfa1`, etc.)
- Radio chip (`Rf230`, `Rf212`, or `Rf231`)
- Compiler (`IAR` or `GCC`)

NOTE Not all combinations make sense for a given platform. Makefiles are provided only for supported configurations.

The image names for pre-built reference applications follow similar a naming convention (see platform specific chapters in [1]).

1.2.2 Building applications with IAR Embedded Workbench

IAR Embedded Workbench can be used to develop and build BitCloud applications. All reference applications include IAR project files located in the `\iar_projects` subdirectory of the application root directory. IAR projects come complete with a set of configurations, which correspond to the configurations given by low-level makefiles.

IAR Embedded Workbench GUI allows the user to select an appropriate configuration from the list of available configurations and to modify any given configuration. For details on compilation and editing configurations refer to IAR Embedded Workbench documentation.

As mentioned above, BSP, PDS, CS stack components, and drivers are compiled with the application. For convenience reasons, source files for these components are included in the IAR projects, so they are effectively a part of the application.

Note that additional configuration parameters including ConfigServer parameters are still contained in the `configuration.h` file in the application root directory.

For compilation from the command line with the IAR compiler, makefiles are used in exactly the same way as described in Section 1.2.1.

2 Modifying HAL Walkthrough

2.1 Overview and General Instructions

The Hardware Abstraction Layer (HAL) is the main stack component responsible for interfacing the stack and the application with the hardware. Therefore, porting the stack to a new platform typically requires modifications to HAL. This chapter describes the most common types of HAL modifications and how to perform them.

- Section 2.2 describes configuration of the MCU and the RF chip interface, that is, how to correctly assign pins and RF interrupts when connecting the MCU and the RF chip.
- Section 2.3 describes handling a node's unique ID which is used as the extended address in a ZigBee network. In standard stack configuration UID is read from the external UID chip when the stack is initialized.
- Section 2.4 covers various HAL parameters configuration such as main clocks and fuse bits.
- Section 2.5 describes how to modify the external flash driver to work with custom devices. The flash driver's primary use is in the applications relying on OTAU functionality.
- Finally, Section 2.6 deals with IO pins and external interrupts.

2.1.1 HAL structure

By definition, HAL code is platform dependent, so separate implementation is provided for each microcontroller. Directory structure inside the HAL root folder reflects the division of microcontrollers into families and particular types. For example, ATmega1281 sources reside in `<Stack-root>\Components\HAL\avr\atmega1281`. The directory containing microcontroller-specific code in turn contains subdirectories for specific board or module configurations. For example, for ATmega1281 there are such subdirectories as `rcb230`, `rcb231_212` and `zigBit` corresponding to specific supported platforms where ATmega1281 is used. Common source file may also be found at each directory level.

A separate subdirectory inside HAL root directory contains sources for drivers, namely `<Stack-root>\Components\HAL\drivers`.

A recommended method to make changes to HAL is to use existing directory structure, modifying contents of existing header and source files.

2.1.2 Compiling HAL

Before any HAL mofidications can be used by an application, the HAL component needs to be recompiled separately. Once that is done the user application can be built with new HAL library. Note that all other stack components for which source code is provided (BSP, PDS, CS, and drivers) are compiled with the application.

HAL sources are located in the `<Stack-root>\Components\HAL` directory. Files in the directory include `Makefile` and `Makerules`, which specify HAL compilation options. It is not recommended to change these files. Additional configuration options can be found in the `Configuration` file. Changing HAL options to fit the specifics of a target board is described in Section 2.4.

To compile HAL run the command line in the `<Stack-root>\Components\HAL` directory and execute the following command:

```
make clean all
```

Assuming the parameters in the `Configuration` file are selected correctly, the command will compile a HAL library based on the combined configuration contained in `Configuration`, `Makefile` and `Makerules` files. Once the HAL compilation is complete, application build process will make use of just created HAL libraries.

2.2 MCU/RF Interface

The list below outlines major steps of configuring MCU and RF interface on a custom board:

1. Properly assign pins interconnections including pins for radio chip interrupts and pins enabling the SPI interface operation. This pins are defined in the `halRfPio.h` file using the `HAL_ASSIGN_PIN (name, port, pin)` macros. Note that `name` in pin assignments should never be changed.
2. Modify source code implementing data transmission in the `halRfCtrl.c` file.
3. Configure interrupt vectors.

NOTE

Specific microcontroller pins and ports may not support all the features required by the transceiver interface. To find out which pins are suitable, please refer to the microcontroller's datasheet.

IMPORTANT

BitCloud requires that the data exchange between an MCU and a radio chip is performed through pure SPI. Hence, the MCU pins used for communication with the RF chip shall support (and not emulate) SPI connection.

The following subsections provide detailed guide for different microcontrollers.

2.2.1 ATmega1281 specifics

The HAL sources for ATmega1281 are located in the `HAL\avr\atmega1281` directory. Move to a board-specific folder inside `\atmega1281` that is chosen as the basement for the customized sources. All paths below in this section are given as relative to the directory you have moved to.

Proceed with the following instructions:

1. Modify pin assignments in the `include\halRfPio.h`:
   a. RF pins: `RF_SLP_TR`, `RF_RST`, and `RF_IRQ`. The RCB230 board also requires the `RF_TST` pin to be defined.

For example, on RCB230 the `RF_SLP_TR` pin is set with the `HAL_ASSIGN_PIN(RF_RST_TR, B, 5);` line, while on ZigBit with `HAL_ASSIGN_PIN(RF_SLP_TR, A, 7);`

   b. SPI pin assignments (`SPI_CS`, `SPI_SCK`, `SPI_MOSI`, and `SPI_MISO`) shall not be changed, because ATmega1281 provides only four pins for SPI connection with the radio chip (B0, B1, B2, and B2).

2. Configure the RF interrupt handler in the `src\halMacIsr.c` file (the last routine in the file). Specify external interrupt in the argument of the ISR routine. For example, on ZigBit it is done by the following line:

   ```
   ISR(INT5_vect)
   ```

   The body of the handler can be left unchanged.

3. Modify code for controlling RF IRQ in the `src\halRfCtrl.c` file. Check that appropriate changes are made in the following functions implemented in the file:

   ```
   HAL_ClearRfIrqFlag()
   HAL_EnableRfIrq()
   HAL_DisableRfIrq()
   HAL_InitRfPins()
   HAL_InitRfIrq()
   ```

   If you change the pin used for RF interrupt, you may need to modify MCU registers accordingly. Also note the following:

   a. `HAL_EnableRfIrq()` and `HAL_DisableRfIrq()` functions involve the same MCU register (`EIMSK` on ZigBit, `TIMSK1` on RCB230).

   b. Give a special attention to initialization functions. For instance, on RCB230 as compared to ZigBit the `HAL_InitRfPins()` function contains two additional lines configuring `RF_TST` pin, namely:

      ```
      GPIO_RF_TST_make_out();
      GPIO_RF_TST_clr();
      ```

4. If pin assignments and interrupts are changed, the user must make sure that the new pins involved are not used by stack components (BSP, HAL) and in the application for controlling LEDs, DTR, etc.

5. To apply changes recompile HAL as described in Section 2.1.2.

2.2.2 XMEGA specifics

HAL sources for various XMEGA microcontrollers are located in subdirectories of the `HAL\xmega` directory. Move to a board-specific subdirectory of the folder corresponding to your MCU. For example, if you use ATxmega256a3, move to the `HAL\xmega\atxmega256a3\stk600` directory. All paths below in this section are given as relative to the directory you have moved to.

Proceed with the following instructions:

1. Modify pin assignments in the `include\halRfPio.h`:

   a. RF pins: `RF_SLP_TR`, `RF_RST`, and `RF_IRQ`.

   b. SPI pins: `SPI_CS`, `SPI_SCK`, `SPI_MOSI`, and `SPI_MISO`.

      For example, you may want to change the port for connection from `C` to `D`, thus the line `HAL_ASSIGN_PIN(SPI_CS, D, 4);` will replace `HAL_ASSIGN_PIN(SPI_CS, C, 4);` and so on.

2. In case you have changed the port of the `RF_IRQ` pin:

a. In `\HAL\xmega\Makefile` in the `RF_PORT_LETTER=C` line replace `C` with the desired port name. For example, `RF_PORT_LETTER=D`.

b. In `src\halMacIsr.c` in the `ISR(PORTC_INT0_vect)` line put `PORT<port_letter>` instead of `PORTC`. For example, `ISR(PORTD_INT0_vect)`.

c. In `src\halMacIsr.c` modify line

```
ISR(TCC1_CCB_vect)
```

and in `..\common\src\halAppClock.c` modify lines

```
#if defined(RF_PORT_LETTERC)
ISR(TCC1_OVF_vect)
#elif defined(RF_PORT_LETTERD)
ISR(TCD1_OVF_vect)
#endif
```

so that the routines they declare use the same port letter. That is, change the line from `src\halMacIsr.c` to

```
ISR(TC<RF_port_letter>1_CCB_vect)
```

and replace the above conditional expression from `..\common\src\halAppClock.c` with a single line

```
ISR(TC<RF_port_letter>1_OVF_vect)
```

3. In case you have changed the pin of the `RF_IRQ` pin assignment :

   a. In `src\halRfCtrl.c` in the `HAL_InitRfIrq()` function in the following lines

   ```
   RF_PORT.PIN2CTRL = PORT_ISC0_bm | PORT_OPC_PULLDOWN_gc;
   ...
   RF_PORT.INT0MASK = PIN2_bm;
   ```

   put `PIN<RF_pin_number>` instead of `PIN2`.

4. In case you have changed the port for SPI pins:

   a. In the `include\halRfSpi.h` file replace all occurrences of `SPIC` with `SPI<port_letter>`. For example, replace with `SPID` if the port has been changed to `D`.

   b. In the `src\halRfSpi.c` file replace all occurrences of `SPIC` with `SPI<port_letter>` For example, replace with `SPID` if the port has been changed to `D`.

5. If pin assignments and interrupts are changed, the user must make sure that new pins involved are not used by stack components (BSP, HAL) and in the application for controlling LEDs, DTR, etc.

6. To apply changes recompile HAL as described in Section 2.1.2.

NOTE For SPI pin assignments you can only change the port, but not the pin numbers. The port must be changed for all SPI pins at once.

2.2.3 SAM3S specifics

HAL sources for SAM3S are located in the `HAL\cortexm3\at91sam3s4c` folder. The stack supports only the SAM3S-EK board by default; therefore move to the `HAL\cortexm3\at91sam3s4c\sam3sEK` folder.

Proceed with the following instructions:

1. Modify RF pin assignments in the `include\halRfPio.h` file:
   a. Skip to the definition of `IRQ_RF_PIN` and set it to the pin number for the RF interrupt. For example:
      ```
      #define IRQ_RF_PIN         17
      ```
   b. Skip to the definition of `IRQ_RF_PORT` and set it to the port letter for the RF interrupt. For example:
      ```
      #define IRQ_RF_PORT        IRQ_PORT_A
      ```
2. Modify SPI pin assignments in the `include\halRfPio.h` file:
   a. Skip to the lines:
      ```
      #define SPI_RF_NPC         PIO_PB2
      ...
      #define SPI_RF_SLP         PIO_PA15
      ...
      #define SPI_RF_RST         PIO_PA18
      ```
   b. Set `SPI_RF_NPC`, `SPI_RF_SLP`, and `SPI_RF_RST` to values of the following format:
      ```
      PIO_P<port_letter><port_number>
      ```
      For example, to specify a GPIO on port `B` under number `11` use the `PIO_PB11` value.
3. If pin assignments and interrupts are changed, the user must make sure that new pins involved are not used by stack components (BSP, HAL) and in the application for controlling LEDs, DTR, etc.
4. To apply changes recompile HAL as described in Section 2.1.2.

2.3 Extended Address Assignment

2.3.1 Assigning the extended address

For correct network operation all devices in a ZigBee network must have unique 64-bit extended addresses (also called IEEE addresses). The extended address value is stored in the `CS_UID` parameter of the Configuration Server and assigned each time the device is powered on.

If the parameter is not set to a non-zero value in the `configuration.h` file, the stack tries to load it from the internal EEPROM. If after this the external address still equals 0, the stack attempts to read it from an external UID chip with the help of the `HAL_ReadUid()` function. If the stack fails to read the value, the extended address will equal 0 and the device will not be able to join any network. In this case the application must set the extended address at run time explicitly.

2.3.2 Options for specifying the extended address

The user can rely on external resources to set the extended address at run time (for example a UID chip or commands from a host MCU) or assign it at compile time in the `configuration.h` file of the application. The only rule is that the extended address must be set to a non-zero value before a device joins the network, and all devices in the same network must have different addresses.

There are four options for specifying the extended address:

1. Duplicate the Atmel development board setup on a custom board, that is, use the same UID chip connected to the MCU in the same way as on one of the supported boards.
2. Implement the `HAL_ReadUid()` function for your custom platform.
3. Assign a non-zero value to the `CS_UID` parameter in `configuration.h`.
4. Set the `CS_UID` value at run time before the device performs network start.

NOTE If the `CS_UID` parameter is specified at compile time, then the application shall be compiled separately for each device.

2.3.3 The `HAL_ReadUid()` function

The `HAL_ReadUid()` function is intended to read the value of the external address from an external UID chip. For each supported platform configuration this function has separate implementation. If you would like the stack to read the UID value from an external UID chip, you may need to modify this function implementation to fit your custom hardware configuration.

`HAL_ReadUid()` is defined in the `halUid.c` file located in the board-specific directory (see Section 2.1.1). For example, for ZigBit this file is located on the `<Stack-root>\Components\HAL\avr\atmega1281\zigBit\src\halUid.c` path. In some implementations the `HAL_ReadUid()` function only returns the value obtained earlier by the `halReadUid()` function inaccessible from outside HAL. In such cases do not change `HAL_ReadUid()` implementation, but modify `halReadUid()`.

2.4 Configuring HAL Parameters

HAL configuration parameters are declared in the `Configuration` file located in `<Stack-root>\Components\HAL`.

Most of the parameters make sense only for some of the platforms. Parameters values are specified for those platforms where there is a choice among multiple possible values and, which require different code to be executed. That is why possible values for the parameters are used as defines for conditional compilation of the source code. For example, the lines below add the value of the `HAL_FREQUENCY` as a macro, provided the parameter itself is defined:

```
ifdef HAL_FREQUENCY
  CFLAGS += -D$(HAL_FREQUENCY)
endif
```

Table 2-1 describes major configuration parameters.

Table 2-1. HAL Configuration Parameters

Parameter	Description
BUILD_CONFIGURATION	In order to debug HAL set this parameter to `DEBUG`, otherwise to `RELEASE`.
PLATFORM	Platform selection
HAL	Microcontroller selection. Some boards support several MCU types so a particular MCU shall be specified for them. The parameter is used in HAL Makefile (see Section 2.1.2) to point to a specific directory containing files for this type of the MCU.
HAL_FREQUENCY	Frequency at which the MCU will operate.

Parameter	Description
HAL_CLOCK_SOURCE	Main clock source for the MCU core.
HAL_ASYNC_CLOCK_SOURCE	Clock source for sleep timeouts.
RF_EXTENDER	Radio extender type. The information is used for connection between the MCU and the radio chip
HAL_USE_USART_ERROR_EVENT	Set to `True` to enable USART error callbacks.
HAL_RF_RX_TX_INDICATOR	Enables/disables support of Rx/Tx indication on the RF chip via DIG3/DIG4 pins.
HAL_ANT_DIVERSITY	Turning on/off the antenna diversity feature.

Table 2-2. HAL Configuration Parameters and Platforms

Platform	HAL	HAL_FREQUENCY	HAL_CLOCK_SOURCE	HAL_ASYNC_CLOCK_SOURCE	RF_EXTENDER
ZigBit	ATMEGA1281 ATMEGA2561	HAL_4MHz HAL_8MHz	n/a	n/a	n/a
Raven	n/a	HAL_4MHz HAL_8MHz	n/a	n/a	n/a
USB Dongle	n/a	n/a	n/a	n/a	n/a
STK600	ATXMEGA128A1 ATXMEGA256A3 ATXMEGA256D3 ATMEGA128RFA1 (1)	HAL_4MHz HAL_8MHz HAL_12MHz HAL_16MHz HAL_32MHz	CRYSTAL_16MHz RC_INTERNAL_2MHz RC_INTERNAL_32MHz	RC_ULP RC_32K CRYSTAL_32K	n/a
SAM-7X EK	n/a	n/a	n/a	n/a	n/a
RCB	n/a	HAL_4MHz HAL_8MHz	n/a	n/a	n/a
AVR32 EK1105	n/a	n/a	n/a	n/a	n/a
SAM3S EK	n/a	HAL_4MHz HAL_8MHz HAL_12MHz HAL_64MHz	CRYSTAL_12MHz RC_INTERNAL	RC_ASYNC CRYSTAL_ASYNC	n/a
REB CBB	n/a	HAL_4MHz HAL_8MHz HAL_12MHz HAL_16MHz HAL_32MHz	RC_INTERNAL_2MHz RC_INTERNAL_32MHz	RC_ULP RC_32K CRYSTAL_32K	REB230 REB231 REB212

Notes: 1. For ATMEGA128RFA1 valid HAL_FREQUENCY values are HAL_4MHz and HAL_8MHz only, HAL_CLOCK_SOURCE and HAL_ASYNC_CLOCK_SOURCE values are ignored.

Table 2-2 indicates availability of parameters that are not valid for all platforms as well as supported values for the parameters. Note that if a parameter is not available on a given platform, then it just shall not be specified in the `Configuration` file. Such parameters are set to a fixed value in `Makerules`.

Parameters not mentioned in Table 2-2 shall be set for all platforms except for the parameters that are unique for a particular platform (see Table 2-3).

Table 2-3. Parameters Unique for a Particular Platform

Platform	Parameter	Values
ZibBit	HAL_TINY_UID	TRUE to read UID from TinyA13 MCU FALSE to read UID from DS2411
	HAL_USE_AMPLIFIER	TRUE for ATZB-A24-UFL/U0 devices FALSE for other ATZB devices
RCB	PLATFORM_REV	RCB_ATMEGA128RFA1, RCB230_V31, RCB230_V32, RCB230_V331, RCB231_V402, RCB231_V411, RCB212_V532

2.5 Flash Driver Modification

2.5.1 Modifying drivers when porting BitCloud

Drivers implement logic of higher level than serial interfaces and can make use of certain interfaces while communicating with external devices. A driver usually implements a specific protocol over the serial connection. When porting BitCloud to a custom platform, the user may need to modify drivers that will be used by the stack and the application. Necessary modifications include both proper pin assignments and changes of the source code of the driver.

The sources for drivers are located in the `<Stack-root>\Components\HAL\drivers` directory.

2.5.2 The Flash driver and OTAU

The Flash driver is intended to exchange data with an external Flash memory device during the Over-the-Air Upgrade process. OTAU functionality is enabled on a single device by the OTAU client cluster. Upon receiving pieces of a new firmware image the OTAU client cluster transfers these pieces to the external Flash device, employing the Flash driver API.

The Flash driver public API is defined in the `ofdExtMemory.h` header file located in `<Stack-root>\Components\HAL\drivers\include`.

BitCloud provides implementation of the Flash driver for AT25F2048 and AT45DB041 devices. A custom board may not use supported Flash devices or external Flash devices at all, but rather store application images in some different way. The only requirement for OTAU is to implement the Flash driver public API.

The easiest way to build a driver for a custom Flash memory device is to follow the design of the drivers provided by BitCloud as described in Section 2.5.4.

2.5.3 Fake Flash driver

Sometimes OTAU should be tested without sending data to the external Flash device or even on test boards without external Flash. To provide such possibility BitCloud allows using fake Flash driver which implements the same API as a normal driver with functions that respond correctly to the caller but do nothing.

To use the fake Flash driver set `APP_USE_FAKE_OFD_DRIVER` to `1` in the application `configuration.h` file as follows:

```
#define APP_USE_FAKE_OFD_DRIVER 1
```

By default `APP_USE_FAKE_OFD_DRIVER` is present in `configuration.h` and set to `0`.

2.5.4 Implementing the Flash driver for a custom Flash memory device

2.5.4.1 Changing build configuration

BitCloud allows switching between different Flash devices (provided the devices are supported) in the `configuration.h` file of the application. To support this feature:

1. In case you use makefiles to build the application, in all low-level makefiles that enable OTAU under the `\makefiles` directory:
    a. Add the definition of the custom Flash device name to the `DEFINES` section. For example: `-D_CUSTOM_EXT_FLASH_NAME`. Use this name in source files to mark the code corresponding to your custom Flash.
    b. Add paths to custom source code files to the `SRCS` section.
2. In case you use IAR Embedded Workbench, modify project settings in the IDE:
    a. Add the definition of the custom Flash device name as a define. Use this name in source files to mark the code corresponding to your custom Flash.
    b. Add paths to custom source code files.
3. In `configuration.h` comment lines containing the `EXTERNAL_MEMORY` definition and add the following line:

```
#define EXTERNAL_MEMORY CUSTOM_EXT_FLASH_NAME
```

2.5.4.2 Flash driver parameters and pin assignments

1. Place your custom Flash driver parameters in `ofdMemoryDriver.h` located under `\Components\HAL\drivers\OFD\include` before lines

```
#else
  #error 'Unknown memory type.'
#endif
```

as follows:

```
#elif defined(CUSTOM_EXT_FLASH_NAME)
  //Custom parameters go here
```

2. Configure the pin named `EXT_MEM_CS` at the bottom of `ofdMemoryDriver.h` by modifying the line of type

```
HAL_ASSIGN_PIN(EXT_MEM_CS, <port_letter>, <pin_number>);
```

corresponding to your MCU.

2.5.4.3 Where to place source code

Consider all files mentioned in this section to be located at the following path:

```
<Stack-root>\Components\HAL\drivers\OFD\src
```

Flash driver public API is implemented in the `ofdCommand.c` file. This file should not be changed, because it only provides high-level implementation independent of a particular external Flash device and relying on a number of functions defined separately for each device type.

`ofdAt25fDriver.c` and `ofdAt45dbDriver.c` contain specific source code for AT25F2048 and AT45DB041 devices, accordingly. The source code for a custom external Flash device should also be put to a separate file based on one of these two files.

NOTE

Standard driver implementation uses the SPI interface for communication with an external Flash memory device. A custom driver can be connected to the MCU through a different interface. In this case use the HAL API corresponding to this interface to communicate with the Flash memory device.

2.6 Interfaces and External Interrupts

2.6.1 Modifying interfaces

For each platform BitCloud provides API for certain IO interfaces. The set of supported interfaces depends on the MCU type. Table 2-3 indicates interfaces for which API is provided in BitCloud. The HAL component registers all necessary interrupt vectors binding them to particular MCU pins.

The general guidelines for the user who needs to modify interface implementation are

- To follow existing design
- To use already registered interrupt vectors
- To modify existing HAL routines containing interrupt vectors' implementation

Table 2-4. Interfaces Supported in BitCloud on Different MCUs

Interface	ATxmega256A3/D3	ATmega1281	ATmega128RFA1	SAM3S	SAM7X	UC3
UART	x	x	x	x	x	x
USART	x	x	x	x	x	x
DTR		x				
SPI	x	x(1)	x		x	
TWI		x	x		x	
ADC		x	x		x	
WDT		x	x		x	
IRQ	x	x	x	x	x	x
SLEEP	x	x	x			
GPIO	x	x	x	x	x	x
PWM		x	x			
USB	n/a	n/a	n/a	x	x	
EEPROM	x	x	x	x(2)	n/a	n/a
1-wire		x(3)			x	

Notes:
1. UART operating in the SPI mode
2. Emulated in Flash
3. Implemented in software

NOTE

For SPI, TWI, and 1-wire only master mode is supported, except for ATmega128RFA1. On ATmega128RFA1 slave mode for SPI is also supported.

2.6.2 Using external interrupts

To register a handler for an external interrupt, which is a callback function executed when the interrupt occurs, use the `HAL_RegisterIrq()` function. The function binds the provided handler to a specific pin and a signal type. The application cannot use a pin already occupied by the stack.

The API is available not for all platforms. On those platforms, for which the API is unavailable, the user has to write platform-specific code to add a handler manually for a particular interrupt. For more detail and to explore other API functions refer to [3].

3 Implementing a Custom BSP

3.1 Overview

The Board Support Package (BSP) is the software component that provides application interfaces for controlling board peripherals, that is, buttons, LEDs, etc. BSP composition depends on the hardware development or evaluation board which is targeted by the SDK (see Table 3-1). Since custom board periphery can vary significantly from the evaluation board, it is quite common for the user to implement a custom BSP. This chapter describes the main steps in the process.

Table 3-1. BSP APIs Implemented for Different Boards

API	MeshBean	STK600	SAM7X-EK	SAM3S-EK	RCB	AVR32 EK1105	REB CBB
LEDs	x	x	x	x	x	x	x
Buttons	x	x				x(1)	x
Sliders	x						
Temperature Sensor	x						
Light Sensor	x						
Joystick			x				

Notes: 1. On AVR32 EK115 buttons API is supported for touch buttons only.

3.2 Disable Existing BSP

To disable standard BSP which comes with BitCloud set the `APP_DISABLE_BSP` constant to 1 in the `configuration.h` file of the application. By default `APP_DISABLE_BSP` is defined and set to 0, thus a BSP implementation is enabled.

Disabling of BSP replaces implementations of all BSP API functions with stubs, which generally do nothing and return the success status when called. More precisely, if BSP is disabled:

- Functions supporting LEDs do nothing.
- Functions supporting buttons do nothing (while a real function eventually invokes the callback).
- Functions supporting sensors invoke callbacks which report zero values.
- Functions supporting sliders return zero as a slider value.
- Functions supporting joysticks do nothing.

With such empty BSP the stack can run on any custom board with supported MCU and RF chip, provided the HAL component has been modified and compiled accordingly. It should be also noted that this does not break any sample application as they can be still be safely compiled with the BSP disabled.

3.3 The BSP Structure

BSP sources for a particular platform are located in a corresponding folder inside the `<Stack-root>\Components\BSP` directory. Like all other stack components the BSP operation is managed by the BSP task handler, which is a function named `BSP_TaskHandler()`. The BSP task handler is called by the stack to process pending BSP tasks. For details on task management in BitCloud refer to [4].

Files required by typical BSP implementation are shown in Table 3-2.

Table 3-2 A BSP Implementation's File Structure

File(s)	**Comment**
`\include\bspTaskManager.h`	Defines BSP task flag and enumerates peripherals.
`\src\bspTaskManager.c`	Contains implementation of the BSP task handler.
Header and source files implementing support of a particular peripheral	The files should use HAL API for communication with a peripheral through a particular serial interface.

3.4 Posting Tasks

The stack keeps track of a special bit indicating whether there is a task associated with the BSP. Posting a task means turning this bit to 1. This will force the stack to call the BSP task handler. To post a task for BSP call `SYS_PostTask(BSP_TASK_ID)`.

A typical BSP implementation involves a special variable in which each bit is associated with the presence of task(s) related to a particular peripheral. A convenience function can be used to post tasks from a given peripheral.

For example, consider a board that has buttons, a light sensor, and a temperature sensor. First, declare an enumeration of peripherals defined in `bspTaskManager.h` as follows

```
enum
{
  BSP_BUTTONS      = (uint8_t)1 << 0,
  BSP_LIGHT        = (uint8_t)1 << 1,
  BSP_TEMPERATURE  = (uint8_t)1 << 2,
};
```

Posting a task from BSP can be performed conveniently via a special function:

```
extern volatile uint8_t bspTaskFlags0;
INLINE void bspPostTask0(uint8_t flag)
{
  bspTaskFlags0 |= flag;
  SYS_PostTask(BSP_TASK_ID);
}
```

`bspTaskFlags0` is a variable holding the task flags. If it equals 0, then there are no BSP tasks. The `bspPostTask0()` function can be called upon certain events like pressing or releasing a button in the following way:

```
bspPostTask0(BSP_BUTTONS);
```

3.5 Implementing the BSP Task Handler

The BSP component must implement its task handler which is a function named `BSP_TaskHandler()`. The stack calls this function whenever there is a task to be executed by BSP.

Provided that task posting is implemented as described in the previous section, the task handler can check whether there is a task associated with each given peripheral and call the corresponding handler.

For example, given a board containing buttons, a light sensor, and a temperature sensor the BSP task handler might look like this:

```
void BSP_TaskHandler(void)
{
  if (bspTaskFlags0 & BSP_BUTTONS)
  {
    bspTaskFlags0 &= (~BSP_BUTTONS); //Clear the buttons' flag
    bspButtonsHandler(); //Call a special handler
  }
  ... //Processing other peripherals in exactly
      //the same manner
}
```

3.6 Setting Build Configuration

The BSP component is compiled with the application. Paths to BSP sources are already given in applications makefiles, and the whole BSP is included in the IAR projects. Application configuration also refers to `Makefile` and `BoardConfig.h` files residing in the BSP root folder. These files include build configuration corresponding to all supported boards.

Typically a user would use one of the predefined configurations modifying it to fit the needs of the application. Generally `Makefile` should not be changed, but `BoardConfig.h` can be easily altered by commenting out unnecessary lines referring to peripherals that are not present on the custom board or are not used by the application.

When implementing a totally new configuration corresponding to a custom board, consider changes in both `Makefile` and `BoardConfig.h`.

4 Porting Sample Applications

Most of the sample applications are available not for all platforms. However, application source code is designed to be independent of the platform by eliminating conditional compilation related to the platform type. Thus the source code can be easily ported to any platform. Actually, only build configuration and ConfigServer parameter require modifications to make a sample application work on the platform it does not support.

To port a sample application to another development board for which the application is not provided, the user shall

- Take the source code of the application for the source platform
- Add makefiles or IAR build configurations from any sample application provided for the destination platform
- Modify node and network parameters along with hardware-related options in the `configuration.h` file
- Build the application, program devices, and test the application

This simple scenario is described in detail in Section 4.2, while Section 4.1 provides an example step-by-step tutorial on porting the LowPower application from ZigBit to XMEGA.

Porting a sample application to a custom board is not that much different. If the HAL and BSP stack components have been modified as described in chapters 2 and 3, then an application available for the your source development board should work on a custom board as well, perhaps with slight changes like selecting another UART channel (see Section 4.3).

4.1 Tutorial: Porting the LowPower Application

The LowPower application is provided with BitCloud packages for ZigBit and Raven (in BitCloud versions older than 1.12.0). So the user might need to use this demo on another development board and build a custom application using it as a template.

This tutorial deals with a particular case of porting the LowPower application to XMEGA on the STK600 board and compiling the application with makefiles. However, the instructions are significantly generic and might be useful for porting any sample applications to other boards as well.

4.1.1 LowPower application overview

The LowPower application demonstrates deferred message delivery to sleeping end devices via the polling mechanism. The application involves one coordinator node and several end devices. After joining the coordinator, an end device sends sensor information to the coordinator and falls asleep for 10 seconds, then wakes up and repeats the process. The coordinator periodically sends requests to change the sensor type to its children. Messages are buffered on the coordinator and retrieved by the end devices through the polling mechanism. The coordinator is also connected to a PC with a serial link and transfers information received from end devices to UART. This information can be observed in the terminal window, for example, in Windows Hyper Terminal.

4.1.2 Tutorial part 1: prepare application files

The first step you have to do is to prepare application files: source and header files and build configuration files (makefiles or IAR project files). Proceed with the following steps:

1. Copy the `LowPower` folder from a BitCloud for ZigBit package to the `Applications` directory in the BitCloud for XMEGA package root (or whatever other package you have).
2. Move to the copied version of the `LowPower` directory and delete the `makefiles` directory, `Makefile`, and the `linkerSrc` directory.
3. Copy the `makefiles` directory, `Makefile` (located in the root application folder), and the `linkerSrc` directory from any sample application provided with the BitCloud for XMEGA package to the target `LowPower` directory.
4. In the copied `Makefile`, select build configuration corresponding to your hardware (MCU and radio) by uncommenting lines with appropriate `PROJECT_NAME` and `CONFIG_NAME`.

Now you should be able to compile the application by executing the `make clean all` command in the command line in the context of the `LowPower` directory, although this will not produce a valid application image, because your should first configure application parameters in the `configuration.h` file.

4.1.3 Tutorial part 2: configure application parameters

Once you have prepared files for your ported application, it is time to configure application parameters in the `configuration.h` file located in the root application directory (`<SDK-root>\Applications\LowPower`).

4.1.3.1 Configure node parameters

Before a network join, each device shall set the short address, the device type, and the extended address. The LowPower application uses static addressing, which means that the short address shall be specified manually in the `configuration.h` file as the value of the `APP_NWK_NODE_ADDRESS` parameter, and the application shall be compiled separately for each device. The device type is deduced from a short address value: the device with short address `0x0000` becomes the coordinator, while other devices become end devices.

If the UID chip is present on the board, the extended address is assigned automatically (see Section 2.3), otherwise the extended address must be assigned in the `configuration.h` file to the `CS_UID` parameter. For simplicity, consider that the latter case takes place.

Proceed with the following steps:

1. For the coordinator set `APP_NWK_NODE_ADDRESS` to `0x0000`.
2. For other nodes set `APP_NWK_NODE_ADDRESS` to a non-zero value.
3. For all nodes set the `CS_UID` parameter to distinct 64-bit values, for example, `0x1234567890ABCDEFLL`.

4.1.3.2 Configure target network parameters

Configure the channel mask and the extended PAN ID for the target network. These parameters must be the same for all devices. It is assumed that you specify the channel not used by other ZigBee networks around.

1. Set `CS_CHANNEL_MASK` to an appropriate value.
2. Set `CS_EXT_PANID` to an arbitrary 64-bit value.

4.1.3.3 Configure hardware-specific parameters

Hardware-specific parameters include configuration of the UART interface which is used to transfer data from the coordinator to a PC. You shall configure the `APP_INTERFACE` and `APP_USART_CHANNEL` parameters as described below:

1. Make sure the `configuration.h` file contains the following line, informing the application that UART is going to be used.

   ```
   #define APP_INTERFACE APP_INTERFACE_USART
   ```

NOTE For most of the supported platforms the same port can work as both UART and USART port. By default, BitCloud uses it as UART, although some application parameters names may contain `USART`.

2. Set the UART channel by replacing the

   ```
   #define APP_USART_CHANNEL USART_CHANNEL_1
   ```

   line with

   ```
   #define APP_USART_CHANNEL USART_CHANNEL_F0
   ```

4.1.3.4 Compile the application

You need different applications images for all devices in the network: on for the coordinator and one for each end device.

1. Set parameters for the coordinator.
2. Build the application by executing `make clean all` from the command line in the root application directory.
3. Rename the output `LowPower.hex` file to `LowPowerCoord.hex` or any other appropriate name. Otherwise, it will be rewritten during the next compilation.
4. Set parameters for an end device.
5. Build the application for each end device, changing short and extended addresses in `configuration.h` and renaming the image after each compilation.

4.1.4 Tutorial part 3: prepare hardware and program devices

At this point you have ready firmware images of the LowPower application. You shall now prepare hardware and program devices. In steps 4 and 5 below you can enable LEDs and buttons. If you do this LEDs will indicate application activity and the SW0 button can be used to force the coordinator to issue an additional request to switch the sensor type to its children.

1. Connect the radio device to the STK600 board (make sure you have chosen the correct makefile corresponding to the radio device used) to port `C`.
2. On the coordinator, connect RXD/TXD pins to the PF2/PF2 MCU's pins, respectively (because you have specified F0 as a UART channel), to enable USART.
3. (Optional) connect LED0 – LED7 and nearby GND and VTG pins to the MCU's PORTE to enable LEDs.
4. (Optional) connect SW0, SW1, and SW2 pins to the MCU's F0, F1, and F5 pins, respectively, to enable first three buttons.

NOTE You can change pin assignments for buttons in BSP. In case of XMEGA on STK600 the code configuring pins to use with buttons is located in the `bspInitButtons()` function in `<SDK-root>\BitCloud\Components\BSP\ATML_STK600\src\buttons.c`.

5. Program devices with JTAG as described in [1]. Each device must be programmed with a different application image.

4.1.5 Tutorial part 4: observe application operation

You are now ready to test the ported LowPower application:

1. Connect the coordinator device to a COM port on a PC.
2. Launch Windows HyperTerminal. Select `COM1` in the `Connect using` field.
3. In port settings set `Bit per second` to `38400` and `Flow control` to `None`, leaving other parameters unchanged.
4. Power on the coordinator and then end devices.
5. After several seconds you will start receiving messages from end devices with the sensor type and zero values, which will appear in the terminal window. Sensors will report zeros, because there are no sensors on the STK600 board, and the BSP provides empty implementation of the sensors API, which simply returns zero at any time.

If something goes wrong, check that all steps described in the tutorial are performed correctly.

4.2 Porting to another Development Board

Since all reference applications are not available for all platforms, the user may need to port a reference application to a platform that does not support this application. As seen from Table 4-1, only WSNDemo and Blink are included in all BitCloud packages. Fortunately, BitCloud reference applications are almost completely generic, so it takes only few steps to port an application from one platform to another.

Table 4-1. Reference applications.

Application	**Brief description**	**ZigBit**	**megaRF**	**UC3**	**XMEGA**	**SAM7X**	**SAM3S**
WSNDemo	Featured SDK application demonstrating network functionality of software and additional network visualization with WSNMonitor.	x	x	x	x	x	x
Blink	Introduces the simplest application that uses timer and LEDs. When started, the application makes all the LEDs blink synchronously with a certain period	x	x	x	x	x	x
Lowpower	Shows how to collect data from low-power, sleeping devices employing the simplest power management strategy	x					
Peer2peer	Shows how to organize the simplest peer-to-peer link. A simple buffering strategy is employed to avoid byte-by-byte data transfer	x			x	x	
PingPong	Shows how to process multiple, simultaneous data transmissions. Each node is waiting for a wireless message, and then passes it to the next node	x				x	
ThroughputTest	Measures wireless UART bandwidth	x				x	
ZDPDemo	Demonstrates ZDP requests to reveal properties of remote devices					x	

Application source code does not require modifications except the `configuration.h` file containing application parameters. Build configuration can be borrowed from any other reference application provided for the destination board. Makefiles can be used without any modifications, while for IAR projects, the user must specify manually the files included into the project.

4.2.1 Overall porting process

To port an application the user shall consider the following steps:

- Copy the source application folder to the `<SDK-root>\Applications\` directory of the target BitCloud package
- Set hardware-related parameters in the application's `configuration.h` file
- Prepare build configuration: makefiles or IAR projects
- Compile, program devices, and test the application

NOTE

Reference applications use LEDs to indicate the application status and sometimes buttons, for additional actions. If buttons are not supported on the board to which the application is ported, the application will still work correctly, because the buttons API for such board is implemented with stubs, which do nothing.

4.2.2 Application parameters configuration

The user must properly configure hardware-related parameters and the extended address in the `configuration.h` file as described below:

- If the target board does not have a UID chip, from which the stack can extract an extended address value, set the `CS_UID` parameter to a 64-bit non-zero value unique throughout the network (and so build a separate image for each device). Otherwise, the `CS_UID` can equal zero (`0x0LL`)
- Configure the serial interface that should be used by the application, namely:
    - Specify the interface type in the `APP_INTERFACE` parameter, for example, if UART is going to be used:

      ```
      #define APP_INTERFACE APP_INTERFACE_USART
      ```
    - Specify the channel and the mode (master/slave for SPI). For example, to use UART channel F0 on STK600:

      ```
      #define APP_USART_CHANNEL USART_CHANNEL_F0
      ```
    - Note that these parameters can be under an `#ifdef` condition checking what board is used. In this case, specify the target board name or simply remove the condition checking.
- Configure network parameters that may be radio-dependent, namely, `CS_CHANNEL_MASK` and `CS_CHANNEL_PAGE`

The second step (setting parameters for serial connection) can be skipped, if the application does not use the serial interface.

4.2.3 Using Makefiles

Makefiles can be taken from any other reference application available for the board to which the application is being ported. To launch the application in order to demonstrate BitCloud features, makefiles does not require modifications. The user shall just copy the whole `makefiles` directory, the high-level `Makefile`, and the `linkerSrc` directory from any reference application already supported by the target platform to the root folder of the application being ported.

4.2.4 Using IAR projects

The easiest way to get IAR project files for the ported application is to copy and modify IAR project files from another application for the target platform as the following steps describe:

1. Replace the `iar_projects` folder in the target application folder with the `iar_projects` folder from any reference application available for the target platform (for example, WSNDemo)
2. Rename the .eww file giving it the name of your ported application
3. For each .ewp file:
    a. Open the file in a text editor
    b. Skip to the `<group>` tag containing the `<name>src</name>` field and enumerate all application's source (.c) files. For example,

```
<group>
  <name>src</name>
  <file>
    <name>$PROJ_DIR$/../src/portedApp.c</name>
  </file>
```

```
    <file>
      <name>$PROJ_DIR$/../src/boardAbstraction.c</name>
    </file>
  </group>
```

c. Skip to the `<group>` tag containing the `<name>include</name>` field and enumerate all application's header (.h) files from the `include` directory. For example,

```
  <group>
    <name>include</name>
    <file>
      <name>$PROJ_DIR$/../src/portedApp.h</name>
    </file>
  </group>
```

Once the instructions above are executed, open the .eww file via IAR Embedded Workbench to browse and choose among available configurations.

4.2.5 Further customization

This section briefly describes some common steps for customizing a sample application to fit custom user requirements.

4.2.5.1 Adding and removing files

Put additional source and header files to the `src` and `include` directories of the application, respectively.

If makefiles are used for compilation, all files from these two directories are compiled, so makefiles does not require changes.

If the application is compiled from IAR Embedded Workbench, to compile additional files with the application add them to the IAR project from IAR Embedded Workbench.

To remove a file from the application, delete it from the project in whatever IDE you use and then delete the physical file (from the `src` or `include` directory).

4.2.5.2 Renaming an application

If an application is compiled from IAR Embedded Workbench, the user can safely rename the .eww file in the `iar_projects` and the application folder. But this will not change the name of the output .hex file and names of other output files. Names for output images are specified in `Options > Linker > Output` and `Extra Output` in the IDE.

If an application is compiled from the command line with the help of makefiles, to rename the application specify a different value for the `APP_NAME` variable in the high-level Makefile. But note, that the high-level Makefile resets the value of `APP_NAME` defined in each low-level makefile, so if the application is compiled directly via a low-level makefile, specify the right value for `APP_NAME` in the low-level makefile as well.

To debug a renamed application from AVR Studio, the user also needs to specify the right name of the .elf file in the AVR Studio project file (the .aps file in the root application folder). To do this, open the .aps file in a text editor and modify the `<ObjectFile>` field.

4.3 Porting to a Custom Board

If the reference application is not provided with the SDK for the user's platform, then its sources can be taken from a BitCloud package supporting this application (see Table 4-1). The application should then be ported to the development board supported by the user's platform as described in Section 4.2. Once the application is available on the evaluation board (either it has been ported or is provided in the SDK), the user should modify HAL and BSP as shown in chapters 2 and 3 addressing a custom board configuration.

From that moment, the reference application should work correctly on the custom board. The user may only need to change the UART channel used by the application (see Section 4.2.2) and disable BSP, if buttons and LEDs API is not implemented for the custom board. To disable BSP, in the `configuration.h` file set `APP_DISABLE_BSP` to `1` (see Section 3.2 for detail).

5 References

[1] AVR2052: BitCloud Quick Start Guide

[2] AVR2055: BitCloud Profile Suite Quick Start Guide

[3] BitCloud Stack API Reference

[4] AVR2050: BitCloud Developer's Guide

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: (+1)(408) 441-0311
Fax: (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Milennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chou-ku, Tokyo 104-0033
JAPAN
Tel: (+81) 3523-3551
Fax: (+81) 3523-7581

© 2011 Atmel Corporation. All rights reserved. / Rev.: CORP072610

Atmel®, Atmel logo and combinations thereof, AVR®, AVR® logo and others are registered trademarks or trademarks of Atmel Corporation or its subsidiaries. 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 product 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.