



BLE Software Users Guide

Contents

1	Introduction	4
1.1	Purpose	4
1.2	Scope of this Manual	4
1.3	Acronyms and Terms	5
1.4	References	5
1.5	Additional Information	5
2	Product Overview	6
3	Installation Components	7
3.1	CCES Installation	7
3.1.1	Web Installation	8
3.1.2	Local Installation	9
4	Example Projects	10
4.1	Sensor Examples	10
4.2	Opening Example	10
4.3	Example Layout	11
4.4	Running the example	11
5	Creating New Projects	12
5.1	CCES Project Creation	12
5.2	Run-Time Environment Configuration	12
5.3	Debug Configuration	13
6	Bluetooth Low Energy Software	15
6.1	Overview	15
6.2	Communication Model	15
6.3	Software Organization	16
6.4	Software Use	17
6.4.1	Command and Response	17
6.4.2	Receiving Expected Events	18
6.4.3	Receiving Unexpected Events	19
6.4.4	Data Received in Events	21
6.5	OTP Interface	21
6.5.1	Overview	21
6.5.2	Programmer	22
6.5.3	MAC Address Modifying	22

1 Introduction

1.1 Purpose

This document describes the Bluetooth Low Energy (BLE) Software Pack for CrossCore Embedded Studio® (CCES) . This software package is targeted at the EM9304 Bluetooth Low-Energy chip paired with an support Analog Devices' low-power microcontroller.

1.2 Scope of this Manual

This document describes the EM9304 Bluetooth Low Energy software package and examples. It explains what is included in this pack.

This document is intended for users who want to write software for the EM9304 Bluetooth Low Energy targeted at an Analog Devices' low-power microcontroller. It assumes some familiarity with the processor's Device Family Pack and the C programming language.

Note that this document does not cover the IoTNode Android application that is compatible with the Bluetooth Low Energy. Please see the IoTNode Users Guide for documentation on the Android application.

1.3 Acronyms and Terms

ADI	Analog Devices, Inc.
API	Application Programming Interface
BSP	Board Support Pack
CCES	CrossCore Embedded Studio®
CMSIS	Cortex® Microcontroller Software Interface Standard
DFP	Device Family Pack
HRM	Hardware Reference Manual
NoOS	No Operation System
RTE	Run-Time Environment

1.4 References

1. CrossCore Embedded Studio® (CCES) [<http://www.analog.com>]
2. ARM CMSIS PACK [<http://www.keil.com/cmsis/pack>]

1.5 Additional Information

For more information on the latest ADI processors, silicon errata, code examples, development tools, system services and devices drivers, technical support and any other additional information, please visit our website at www.analog.com/processors.

2 Product Overview

The Bluetooth Low Energy (BLE) Software Pack (ADI-BleSoftware) provides a software framework for the EM9304 Bluetooth Low Energy chip. The software is divided into three layers:

- Framework Layer: Control flow for applications with no operating system.
- Companion Layer: Encoding and decoding of BLE packets.
- Transport Layer: Reading and writing of BLE packets.

The package acts as a library and can be used supported Analog Devices' processors. Examples demonstrating how to use the BLE Software can be found in the Board Support Package for the board being targeted.

3 Installation Components

Before installing the BLE Software Pack, the following should be installed.

- CrossCore Embedded Studio ® 2.6.0
- Device Family Pack for the targeted processor
- Board Support Pack for the targeted board ([**Optional**]: only to get example code)

This software is released in the form of a CMSIS Pack file. CCES will extract the contents of the Pack file into the CCES installation directory. This allows for a clean partitioning of software delivered by ADI and software created by the user. The BLE Software Pack contents (software framework, documentation, etc.) are placed at the following location

- CrossCore Embedded Studio® : `<cces_root>/ARM/packs/AnalogDevices/ADI-BleSoftware/x.y.z`

where **x.y.z** is the installed pack version number. Figure 1 shows the contents that will be placed at this location after the installation has completed.







 Documents	8/2/2017 11:08 PM	File folder
 Host	8/2/2017 11:09 PM	File folder
 Include	8/2/2017 11:09 PM	File folder
 License	8/2/2017 11:09 PM	File folder
 Source	8/2/2017 11:09 PM	File folder
 AnalogDevices.ADI-BleSoftware.pdsc	8/2/2017 3:34 AM	PDSC File

Figure 1. Installation Directory Structure

3.1 CCES Installation

To install a new Pack or update an existing Pack go to CMSIS Pack Manager perspective, shown in Figure 2. If the Pack Manager perspective was not opened previously, the CMSIS Pack Manager icon may not be present on the toolbar as shown below. In that case, the Pack Manager perspective can be opened by clicking *Window Perspective Open Perspective Other Pack Manager*. There are two methods that can be used to install the Pack described below.

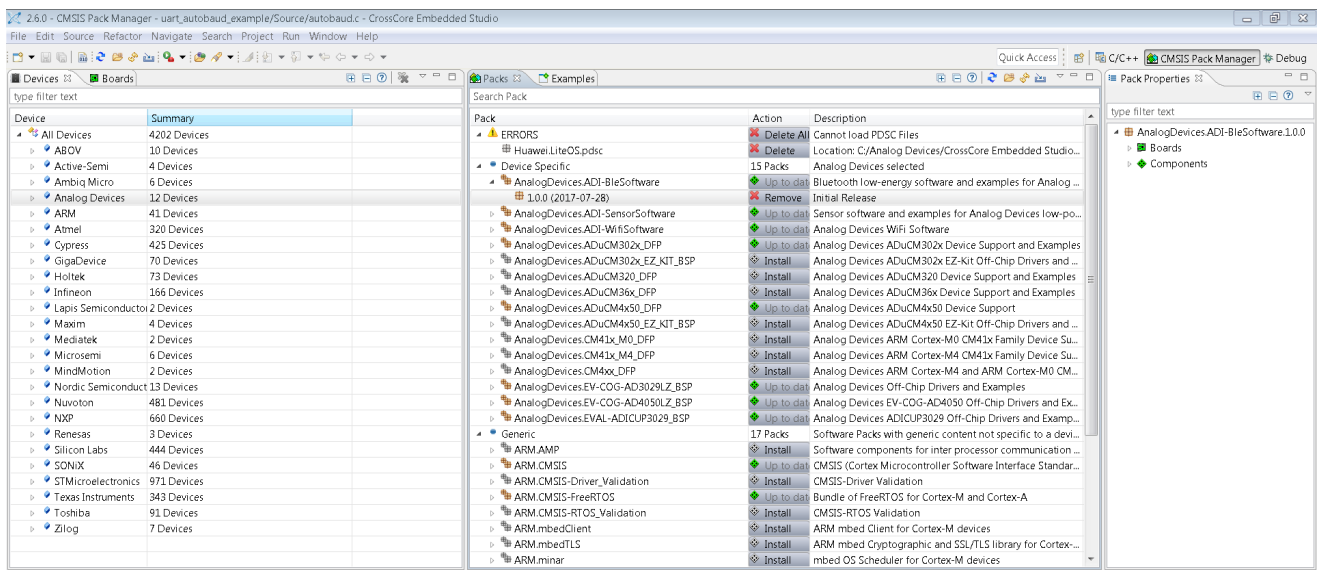


Figure 2. CMSIS Pack Manager Perspective

3.1.1 Web Installation

The Pack can be installed directly from the web using CCES, the user does not need to download the file and open it with CCES. This can be done by first refreshing the CMSIS Pack Manager (the blue arrows in the top left of the *Packtab*). This will display a list of available Pack files as shown in Figure 3. Clicking on the "ADuCM302x Series" will show the Pack in the *Pack* tab as "AnalogDevices.ADI-BleSoftware". Click "Install" and accept the license agreement in order to install the Pack .

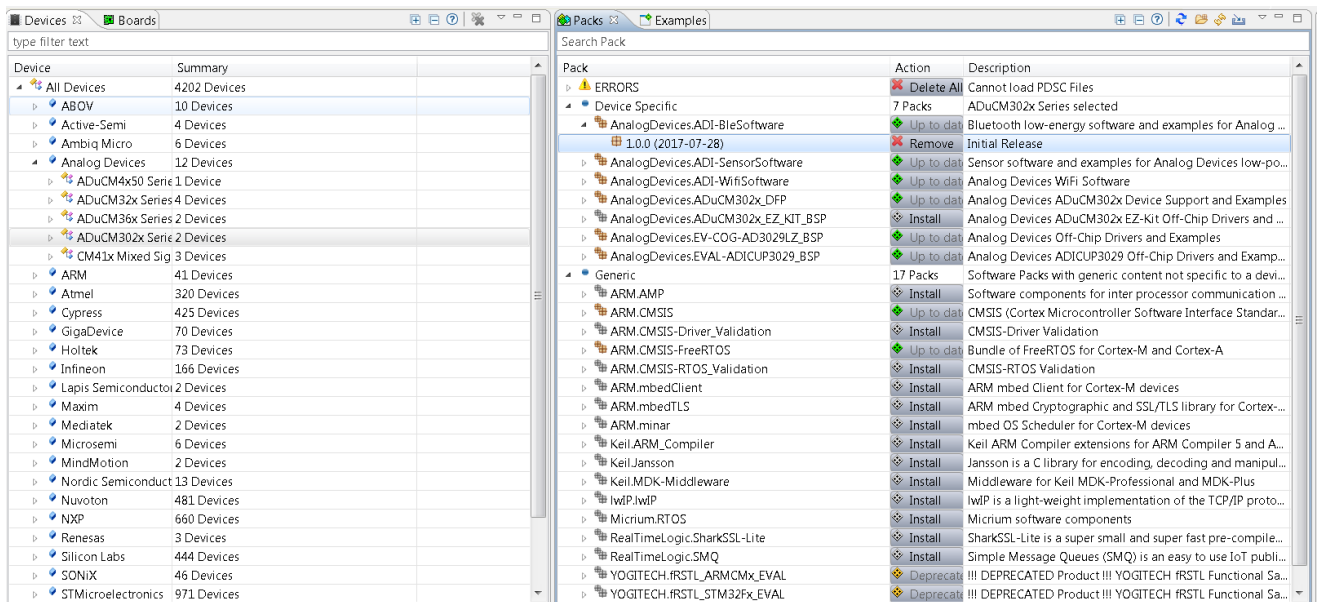


Figure 3. Available Pack Files

3.1.2 Local Installation

If the user has already obtained the Pack file, it can be installed without using the method described above. Click "Import Existing Packs" (the folder icon in the *Pack* tab) and then browse to the Pack file.

4 Example Projects

4.1 Sensor Examples

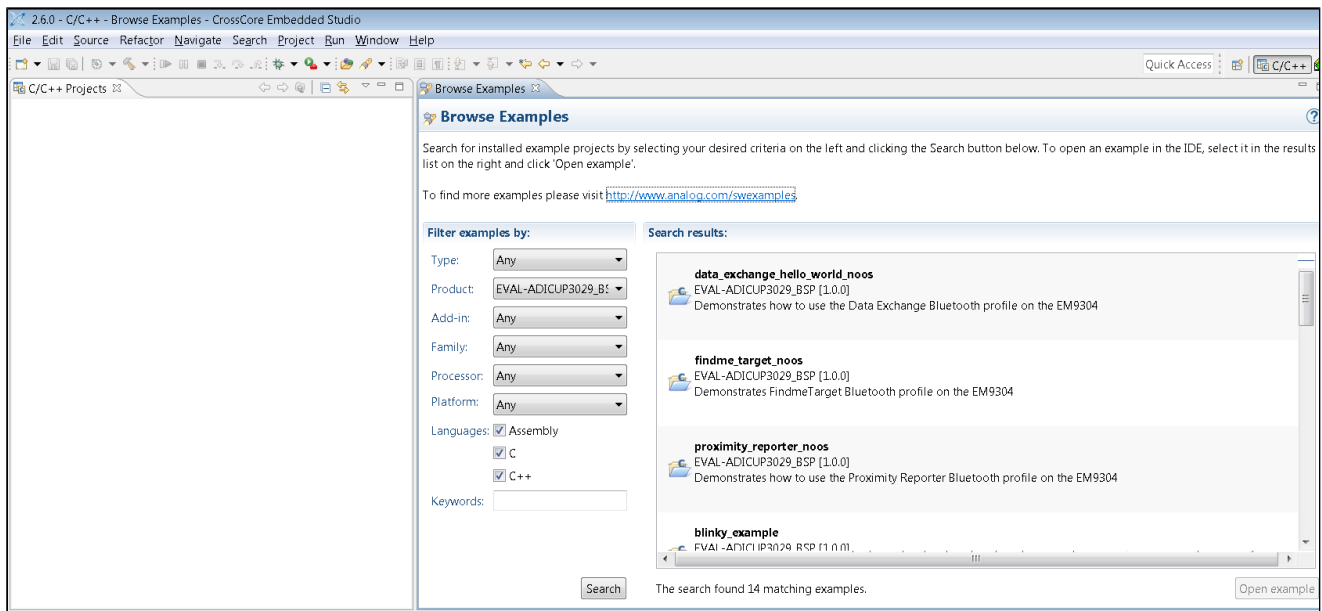
Board Support Packs that support the Sensor Software pack contain the example(s) given below.

- FindMe Target
- Proximity Reporter
- Data Exchange Hello World

All examples work as described in the associated example's readme file. These example applications work with the *IoTNode* android application running in parallel to the application running on the microcontroller. Please refer to *IoT_Node_Users_Guide.pdf* present in the `<cces_root>/ARM/packs/AnalogDevices/ADI-BleSoftware/1.0.0/Documents` folder.

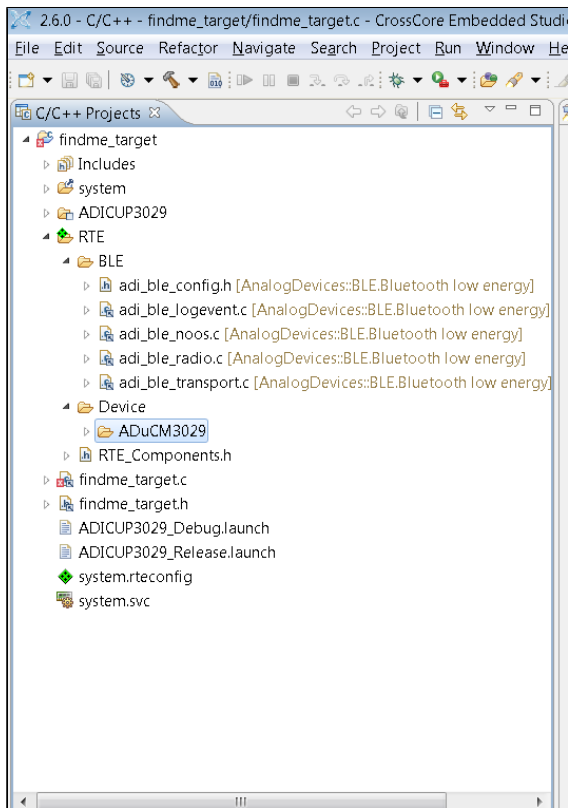
4.2 Opening Example

Examples can be opened by using CCES Browse Examples window as shown below. Double click on the example to open the project.



4.3 Example Layout

The layout of the FindMe Target example is given below. All other examples have a similar layout. Top-level source files are included directly below the project in CCES. The readme, pinmux, and other board specific files are located in the folder with the board name. The RTE folder contains components included from PACK files (device drivers from DFP and Bluetooth Low-Energy software). There are also *.launch* files which are used to run the example with the debugger.



4.4 Running the example

Follow below steps in order to run the project

1. Open the project and build either Debug or Release.
2. Set up hardware as described in the associated project readme file.
3. Click the appropriate *.launch* file and then press "Debug" in the CCES "Run" menu.
4. Press "Run" in the Debug context of CCES (looks like a play button).
5. Open the Android application and connect to the target.

5 Creating New Projects

New projects can easily be created that use the on-chip drivers delivered in the Device Family Pack and the BLE software component in the BLE Software Pack.

5.1 CCES Project Creation

To create a new project in CCES targeting the microcontroller being used, select *File New CrossCore Project* and then provide a name for the project. In the "Processor Type" dialog, select the processor being used and choose a silicon revision (choose "any" if it does not matter). In the "Project configuration" window, select any Add-ins or template code you may want for your new project and then select "Finish".

5.2 Run-Time Environment Configuration

The new project will be added to the "Project Explorer". The software to interface with the on and off-chip peripherals on an Evaluation Board are organized as "components" in the Run-Time Environment. Within the context of this product, a component contains one or more source files, and can also contain a static configuration header file. To add components to the Run-Time Environment for your new project, open the *system.rteconfig* file in the "Project Explorer". The BLE software is located in the "BLE" class and the on-chip drivers for the ADuCM302x are located in the "Device" class. Click these components to add them to the project. Components will have dependencies on other components. If these dependencies are not met, the needed component will be highlighted in yellow. Note that the most minimal project will *Device Startup*, *Device Global Configuration*, and *CMSIS Core*. The components in the "BLE" class will be explained in greater detail in the next section. For more information about the components in the "Device" class, please see the ADuCM302x Device Family Pack.

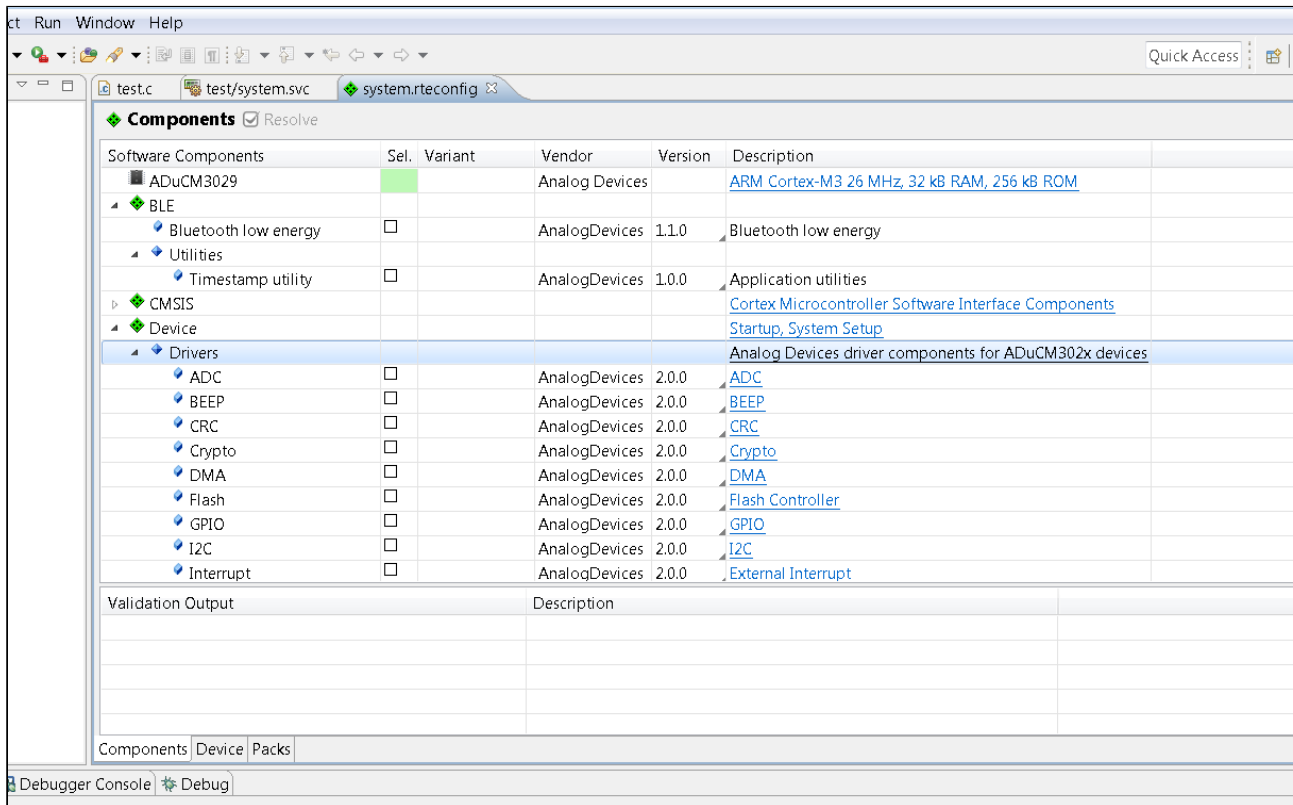


Figure 7. RTE Configuration

5.3 Debug Configuration

If no *.launch* files are available when creating a new project, the user will need to create a debug configuration from scratch. Go to *Run Debug Configurations* to create a configuration. Click "Emulator" and then click the "New" (small white page icon in top left corner). Configure the settings to match Figure 8 if you are using an ADuCM3029 processor with a CMSIS DAP interface.

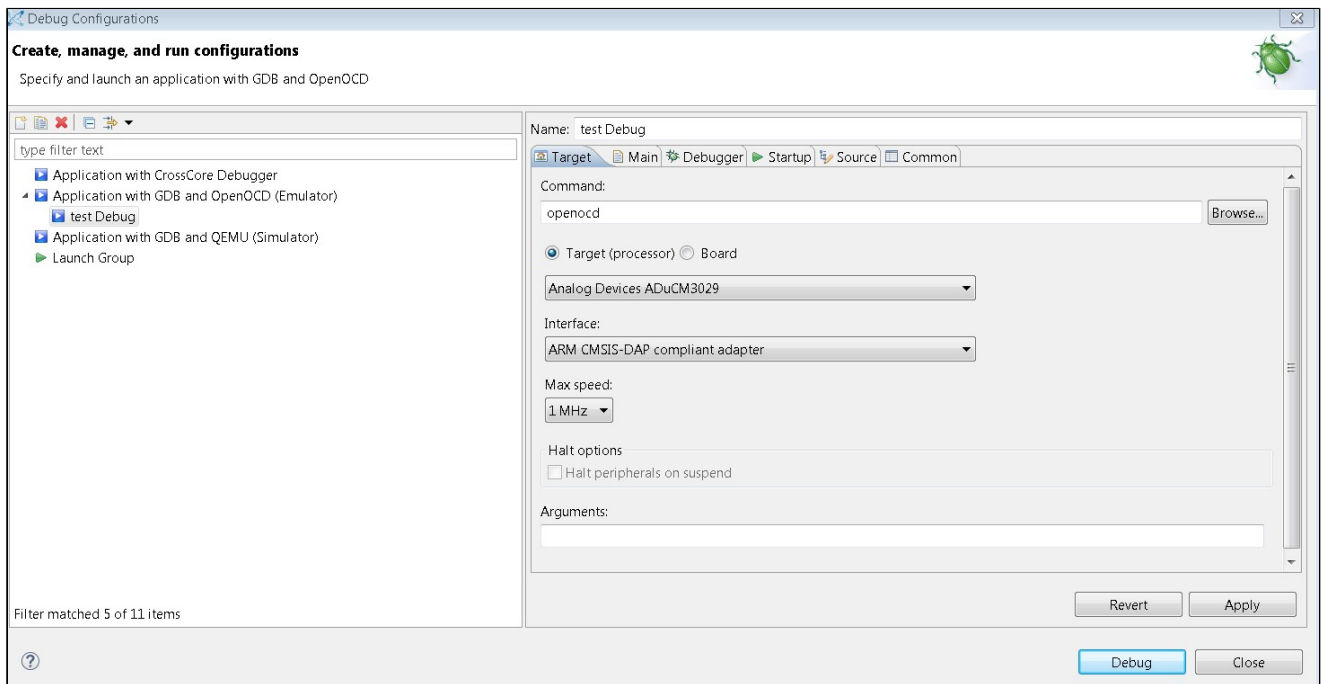


Figure 8. Debug Configuration

6 Bluetooth Low Energy Software

6.1 Overview

The software provided in this package pair an Analog Devices' low-power microcontroller with the EM9304, giving users a simple way to get their applications connected to a host device. The BLE Software Pack contains software that interfaces with the EM9304, and provides the user with a simple, high-level interface for performing Bluetooth Low-Energy (BLE) operations. The BLE software can be incorporated into an application by adding the "Bluetooth low energy" component in the RTE configuration (described in the previous section). This will add the BLE source files, a BLE static configuration, and the necessary include paths to the project. This section describes the organization of the BLE software and how to use it.

Note that this section will cover the software a high-level, please see the Doxygen documentation located in *Documents/html/index.html* for specific details about each API, data structure, enumeration, and macro.

6.2 Communication Model

At a high level, the microcontroller and EM9304 communicate using a command set over a serial interface. After a command is sent from the microcontroller, the EM9304 will always reply with a response. The EM9304 can also issue events to the microcontroller. Unlike responses, events are asynchronous. However, events can be both expected and unexpected. This will be covered in more detail later. Figure 9 illustrates this communication model.

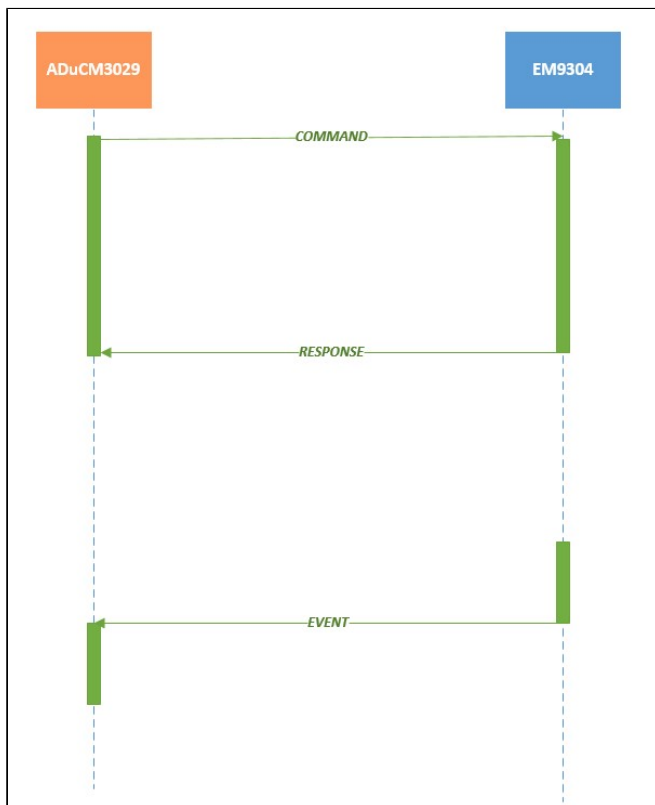


Figure 9. Basic Communication Model

6.3 Software Organization

The BLE software is partitioned into three layers, build on each other and providing different levels of abstraction from the communication model.

Layer	Include	Description
NoOS Framework Layer	<code><framework/noos/adi_ble_noos.h></code>	<i>Control flow</i> This layer is used to manage control flow by blocking until specific responses or events are received. The layer will also handle unexpected events by passing them to the application via a callback function.
Companion (Radio) Layer	<code><radio/adi_ble_radio.h></code>	<i>Encoding and decoding of packets</i> This layer deals with the encoding and decoding of packets sent and received from the EM9304.

Layer	Include	Description
Transport Layer	<code><transport/adi_ble_transport.h></code>	<p><i>Writing and reading packets</i></p> <p>This layer implements the serial interface used to communicate with the EM9304.</p>

6.4 Software Use

Applications interact with the companion and framework layer APIs when performing BLE operations. Depending on the mode of operation, the call sequence will vary. This section will explain how to use these APIs for the various modes of operation. Also, the process of receiving data from the EM9304 will vary based on these modes.

There are three modes of operation for the BLE software:

- Command and response
- Receiving expected events
- Receiving unexpected events

6.4.1 Command and Response

The simplest form of communication with the EM9304 is by issuing commands and receiving a response directly after. Applications perform this action by using the APIs located in the Companion Layer. The command-response sequence is completed by only a single function call from the application as shown below. This is a blocking function call, but the response is received immediately from the EM9304.

```
#include <radio/adi_ble_radio.h>

...

ADI_BLE_RESULT eResult;

eResult = adi_radio_XXX(...);
```

Example 1. Command-Response API Call

The function arguments will vary depending on the API, and are described in the Doxygen documentation. If the response packet will contain data the application needs, the API will have a pointer where this data will be stored. The application must allocate the memory, pass the pointer, and wait for the function call to complete successfully before using the data. An example is shown below.

```
#include <radio/adi_ble_radio.h>

...

ADI_BLE_RESULT eResult;
uint8_t aLocalAddress[6u];

// The response will contain the address, which will be stored in aLocalAddress
eResult = adi_radio_GetConnectionAddr(0x0001u, aLocalAddress);

if (eResult == ADI_BLE_SUCCESS)
{
    // aLocalAddress has been filled with the correct data
}
else
{
    // aLocalAddress has not been filled
}
```

Example 2. Data Received in Response

6.4.2 Receiving Expected Events

Some commands require the EM9304 to perform communication with the remote device. These actions take longer for the EM9304 to perform and are not suitable for the command-response mode of operation. In these cases, an event will occur after the command-response sequence has completed. These are referred to as "expected events", since the application has triggered the event by using a specific command. The Framework Layer contains an API that will allow the application to block until that event has occurred. The use of this API is shown below.

```
#include <radio/adi_ble_radio.h>
#include <framework/noos/adi_ble_noos.h>

...
```

```

ADI_BLER_RESULT eResult;

eResult = adi_radio_XXX(...)

if (eResult == ADI_BLER_SUCCESS)
{

    // Command-response passed, wait for a particular event with timeout
    eResult = adi_ble_WaitForEventWithTimeout(...);

    if (eResult == ADI_BLER_SUCCESS)
    {
        // Event happened in less than the timeout specified
    }
}
else
{
    // Command-response failed, the event will never come
}

```

Example 3. Waiting for a Particular Event

The Doxygen documentation will specify if a command API triggers an event to occur.

6.4.3 Receiving Unexpected Events

The EM9304 can issue events to the microcontroller even if the application did not explicitly trigger it with a command. An example of this would be an application running as a Bluetooth peripheral when a connection is initiated, the connection event will be received unexpectedly and asynchronously. Similarly with a disconnect event, the two devices being moved too far apart could trigger a disconnect which would also be received unexpectedly and asynchronously. In a single-threaded model, the application must periodically pass control to the Framework Layer to handle any unexpected events. An API has been provided to perform this action. If an event occurs during the period in which the application has passed control to the Framework Layer, control will be given back to the application via a callback function. The callback function event contains the event code and the callback function argument may contain data received in the event, depending on the event code. The Doxygen documentation contains details on which events pass data through this callback. An example of this flow is shown below.

```

#include <radio/adi_ble_radio.h>
#include <framework/noos/adi_ble_noos.h>

...

static void ApplicationCallback(void * pCBParam, uint32_t Event, void * pArg)
{
    switch (Event)
    {
        case GAP_EVENT_CONNECTED:

            // Application specific connection code
            break;

        ...

    }

}

int main(void)
{
    ADI_BLE_RESULT eResult;

    eResult = adi_ble_Init(ApplicationCallback, NULL);

    ...

    while(1u)
    {
        // This function call will block for a given period of time and dispatch events to the
        // callback
        eResult = adi_ble_DispatchEvents(...);

        // Application specific code

        ...
    }
}

```

```
}  
  
return 1;  
  
}
```

Example 4. Dispatching Events to the Callback

Note that even though the application callback is executed at the thread level, it should be minimal to avoid blocking other events from being received. Commands shall not be issued from the application level. This concludes the discussion on the three modes of operation for the BLE software. Using the software described here, applications can easily interface with the EM9304 and perform specific actions on BLE events.

6.4.4 Data Received in Events

The interface for receiving data from response packets was simple and straightforward - when the command-response API completes, the memory allocated by the application is filled. For events, it is slightly more complicated due to their asynchronous nature. To simplify the application software, the Companion Layer will cache the data received from an event in local memory. The application then calls a getter API to extract the cached data. The getter APIs are located in the Companion Layer and are stylized as `adi_ble_GetXXX`. Single elements are returned directly and larger structures are returned via a pointer input argument.

The cached data is only valid until the next event occurs, since the next event could be the same as the previous event, and replace the previous data. More explicitly, for an unexpected event, the application needs to call the getter function from the application callback. For an expected event, the application needs to call the getter before the next call to `adi_ble_WaitForEventWithTimeout` or `adi_ble_DispatchEvents`.

6.5 OTP Interface

6.5.1 Overview

The EM9304 contains a one-time programmable memory space for applying patches that contain settings related to the physical communication channel, BLE profiles, and more. Writing and reading to and from this memory space is done using the same command-response-event logic just described. The "Programmer Tool" located in *Tools/programmer* will apply a patch that allows the Bluetooth software delivered in the BLE Software Package to function properly. Some boards may apply this patch is applied in manufacturing, so the user may not need to run it. However, the binary and source forms of the programmer are delivered for users who wish to have more advanced control of the OTP memory space.

6.5.2 Programmer

Running the *programmer.hex* file will ensure that the EM9304 is in the correct communication mode in order to use the Bluetooth software. Multiple runs of this program have no effect, and it only needs to be run once. This is done for "factory floor" environments to prevent incidental programming. Please see the "readme" file located in *Tools/programmer* for more details on how to run this program and the expected output.

6.5.3 MAC Address Modifying

One of the settings that is configured using an OTP patch is the MAC address of the EM9304. To allow multiple boards to function in parallel, the user may wish to modify the MAC address. The default MAC address is 00-05-F7-01-41-44. Applications are delivered in source and binary to change the MAC address of one of 5 predefined MAC addresses. These applications must be run **after** the user has run *programmer.hex* once successfully. The following table lists the applications and the MAC addresses they provide.

CCES Project Configuration	Hex File	MAC Address (hex)
Debug	None	00-05-F7-01-41-44
Release	programmer.hex	00-05-F7-01-41-44
ChangeMAC_45	change_mac_45.hex	00-05-F7-01-41-45
ChangeMAC_46	change_mac_46.hex	00-05-F7-01-41-46
ChangeMAC_47	change_mac_47.hex	00-05-F7-01-41-47
ChangeMAC_48	change_mac_48.hex	00-05-F7-01-41-48
ChangeMAC_49	change_mac_49.hex	00-05-F7-01-41-49

Note that the ChangeMAC_XX applications do not have any kind of "protection" from multiple runs, like *programmer.hex* does. This means that the program will always modify the OTP memory space, even if the new MAC address is the same as the current.