Application Note: JN-AN-1174

IEEE 802.15.4 Application Template for JN516x

This Application Note accompanies the software template for applications designed to run on NXP JN516x microcontrollers in IEEE 802.15.4-based wireless networks. This document introduces and details how to use the template.

1 Application Overview

The software template provides a basis for your own application development for JN516x devices in IEEE 802.15.4-based networks. It is designed to streamline your application development and to help you rapidly achieve effective IEEE 802.15.4 applications. You can modify the supplied code to adapt it to your own application needs. Note that the code is relevant to non-beacon enabled networks only.

The supplied code provides the basic framework for getting a network up and running. It includes the necessary function calls for setting up the network, and then transmitting and receiving data. Separate code is provided for the network Co-ordinator and End Devices.

2 Using the Application Template

The IEEE 802.15.4 application template is supplied in the ZIP package for this Application Note. It should be used in conjunction with the

JN516x IEEE 802.15.4 Software Developer's Kit (SDK) [JN-SW-4163]

in order to develop IEEE 802.15.4 applications in the

'BeyondStudio for NXP' Integrated Development Environment (IDE) [JN-SW-4141]

which is described in the *BeyondStudio for NXP Installation and User Guide (JN-UG-3098)*. These resources are available via the Wireless Connectivity area of the NXP web site.

Note: When developing IEEE 802.15.4 applications for JN516x devices, it may also be useful to refer to the example code in the Application Note *802.15.4 Home Sensor Demonstration for JN516x (JN-AN-1180)*.

2.1 Pre-requisites and Assumptions

It is assumed that you have installed BeyondStudio for NXP (JN-SW-4141) and the JN516x IEEE 802.15.4 SDK (JN-SW-4163) on your development PC.

The skeleton application in the Application Note assumes the following:

- You have one device which will act as the PAN Co-ordinator
- You have at least one other device which will act as an End Device
- You will use pre-determined values for the PAN ID and the short addresses (for the PAN Co-ordinator and for the End Device(s))
- The network topology will be a Star network
- The network will be non-beacon enabled (meaning that the PAN Co-ordinator will not transmit regular beacons)
- Short addressing will be used
- Data transfers will be direct transmissions with acknowledgements
- There will be no security implemented

Before starting your IEEE 802.15.4 application development using the supplied template, you should refer to NXP's *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*, which is available from the Wireless Connectivity area of the NXP web site. This User Guide should be consulted throughout your application development.

2.2 Unpacking the Application Note

Unzip this Application Note (JN-AN-1174) into the Application directory of the SDK installation:

<BeyondStudio for NXP installation root>\workspace

where **<BeyondStudio for NXP Installation root>** is the path into which BeyondStudio for NXP was installed (by default, this is **C:\NXP\bstudio_nxp**). The **workspace** directory is automatically created when you start BeyondStudio for NXP.

You should rename the Application Note folder with the name of your project.

2.3 Supplied Files

The application's file structure includes the following folders (depending on the Application Note):

- **AN1174_154_Coord** - contains source files and makefiles for the PAN Co-ordinator
- **AN1174_154_EndD** - contains source files and makefiles for an End Device
- **Common** - contains the config.h header file used for both devices, which defines certain values used in the source code (e.g. PAN ID, short addresses, channels to scan)

The **AN1046_154_Coord** folder contains **Source** and **Build** sub-folders, the contents of which are described below.

2.3.1 Source Folders

The contents of the **Source** folders are as follows:

- **AN1174_154_Coord/Source** - contains the file **AN1174_154_Coord.c** which contains the source code for the PAN Co-ordinator
- **AN1174_154_EndD/Source** - contains the file **AN1174_154_EndD.c** which contains the source code for an End Device

To adapt the skeleton code to your own needs, you may need to modify the above source files.

2.3.2 Build Folders

The contents of the **Build** folders are similar for the two Application Notes, comprising the makefile (**Makefile**) for compilation of the source code for the JN516x microcontroller.

The **Build** folder is also the place where a compilation outputs the resulting binary file.

3 Code Descriptions

This section describes the supplied source code at function level. The sub-sections below describe the code for the PAN Co-ordinator and the code for the End Device. The **config.h** header file is referenced in both source files, as are the following header files: **jendefs.h**, **AppHardwareApi.h**, **AppQueueApi.h**, **mac_sap.h** and **mac_pib.h**.

3.1 Contents of AN1174_154_Coord.c

The entry point from the boot loader into the Co-ordinator application is the function **AppColdStart()** - this is the equivalent of the **main()** function in other C programs. This function performs the following tasks (also illustrated in Figure 1):

1. **AppColdStart()** calls the function **vInitSystem()**, which itself performs the following tasks:
   - Initialises the IEEE 802.15.4 stack on the device
   - Sets the PAN ID and short address of the PAN Co-ordinator - in this application, these are pre-determined and are defined in the file **config.h**
   - Switches on the radio receiver
   - Enables the device to accept association requests from other devices
2. **AppColdStart()** calls the function **vStartEnergyScan()** which starts an Energy Detection Scan to assess the level of activity in the possible radio frequency channels - the channels to be scanned are defined in the file **config.h** along with the scan duration. Initiation of the scan is handled as an MLME request to the IEEE 802.15.4 MAC sub-layer.
3. **AppColdStart()** waits for an MLME response using the function **vProcessEventQueues()** - this function checks each of the three event queues and processes items found. The function uses the function **vProcessIncomingMlme()** to handle the MLME response. This function calls **vHandleEnergyScanResponse()** which processes the results of the Energy Detection Scan - the function searches the results to find the quietest channel and sets this as the adopted channel for the network. The last function then calls **vStartCoordinator()** which sets the required parameters and then submits an MLME request to start the network (note that no response is expected for this request).

4. **AppColdStart()** loops the function **vProcessEventQueues()** to wait for an association request from another device, which arrives as an MLME request (note that the beacon request from the device is handled by the IEEE 802.15.4 stack and is not seen by the application). When the association request arrives, the function **vHandleNodeAssociation()** is called to process the request. This function creates and submits an association response via MLME.
5. **AppColdStart()** loops the function **vProcessEventQueues()** to wait for messages from the associated device arriving via the MCPS and hardware queues.

   - When data arrives in the MCPS queue, **vProcessEventQueues()** first uses the function **vProcessIncomingMcps()** to accept the incoming data frame. Note that **vProcessIncomingMcps()** uses **vHandleMcpsDataInd()**, which calls **vProcessReceivedDataPacket()** in which you must define the processing to be done on the data.
   - When an event arrives in the hardware queue, **vProcessEventQueues()** calls the function **vProcessIncomingHwEvent()** to accept the incoming event. You must define the processing to be performed in this function.

Note: As it stands, the code is only designed to receive data. To transmit data from the PAN Co-ordinator, you must modify the code - a transmission function is provided (see Section 4.6).

The above Co-ordinator set-up process is illustrated in Figure 1 below.

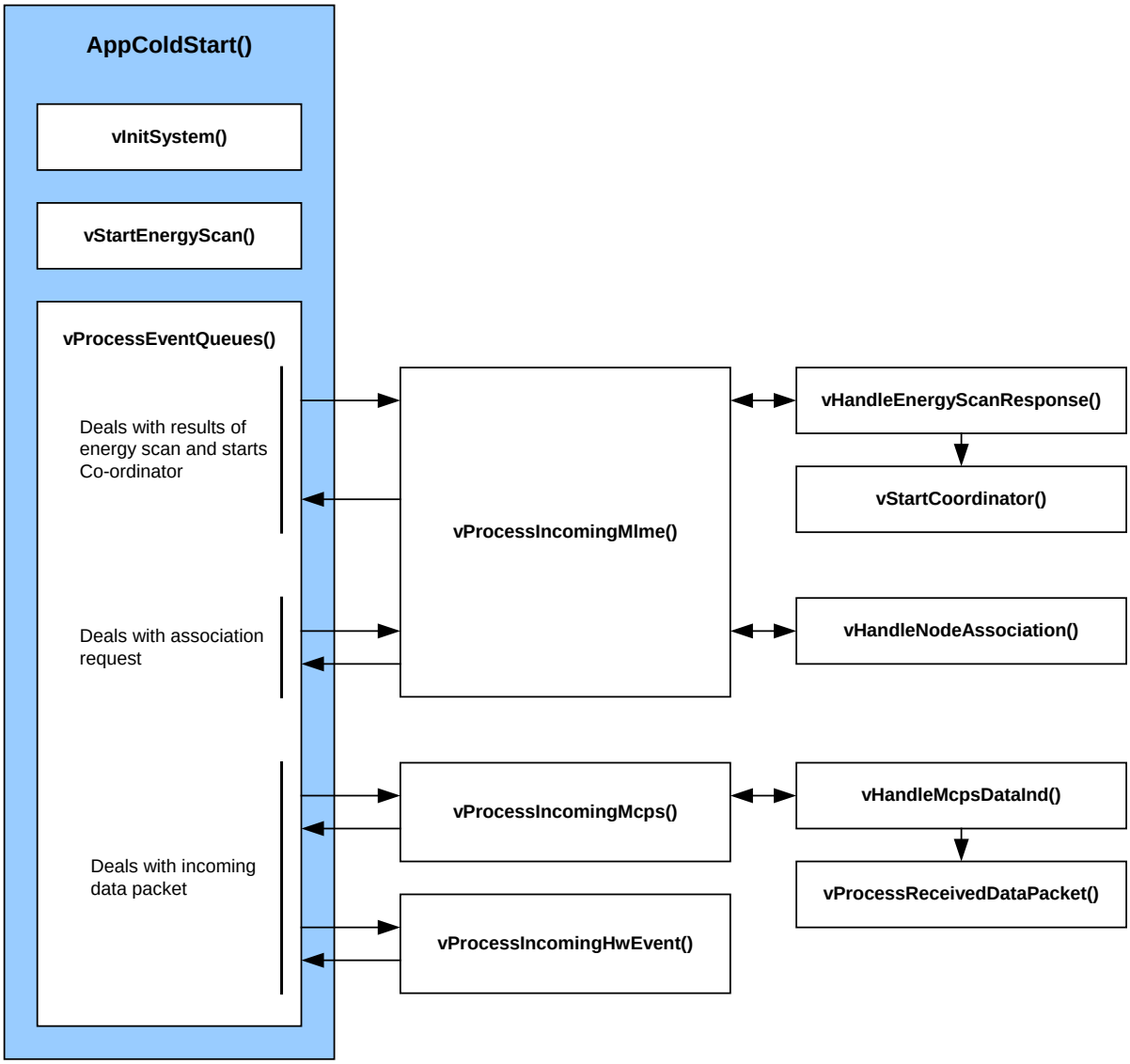

Figure 1: PAN Co-ordinator Set-up Process

3.2 Contents of AN1174_154_EndD.c

The entry point from the boot loader into the End Device application is the function **AppColdStart()** - this is the equivalent of the **main()** function in other C programs. In **AN1174_154_EndD.c**, this function is defined differently from that in **AN1174_154_Coord.c**. For the End Device, it performs the following tasks (also illustrated in Figure 2):

1. **AppColdStart()** calls the function **vInitSystem()**, which initialises the IEEE 802.15.4 stack on the device.
2. **AppColdStart()** calls the function **vStartActiveScan()** which starts an Active Channel Scan in which the device sends beacon requests to be detected by the PAN Co-ordinator, which then sends out a beacon in response - the channels to be scanned are defined in the file **config.h** along with the scan duration. Initiation of the scan is handled as an MLME request to the IEEE 802.15.4 MAC sub-layer.
3. **AppColdStart()** waits for an MLME response using the function **vProcessEventQueues()** which checks each of the three event queues and processes the items it finds. The function uses the **vProcessIncomingMlme()** function to handle the MLME response. This function calls the function **vHandleActiveScanResponse()** which processes the results of the Active Channel Scan:
   - If a PAN Co-ordinator is found, the function stores the Co-ordinator details (PAN ID, short address, logical channel) and calls **vStartAssociate()** to submit an association request to the Co-ordinator - this is handled as an MLME request.
   - If a PAN Co-ordinator is not found (possibly because the Co-ordinator has not yet been initialised), the function recalls **vStartActiveScan()** in order to restart the scan (in which case this process continues as described from Step 2).
4. **AppColdStart()** loops the function **vProcessEventQueues()** to wait for an association response from the Co-ordinator. When the response is received, **vProcessIncomingMlme()** is called, which (provided that the device is in the associating state) calls the function **vHandleAssociateResponse()** to process the response. The last functions checks the association response:
   - If the PAN Co-ordinator has accepted the association, the function puts the device into the 'associated' state.
   - If the PAN Co-ordinator has rejected the association, the function recalls **vStartActiveScan()** to start a search for another PAN Co-ordinator (in which case this process continues as described from Step 2).
5. **AppColdStart()** loops the function **vProcessEventQueues()** to wait for messages from the PAN Co-ordinator arriving via the MCPS and hardware queues.
   - When data arrives in the MCPS queue, **vProcessEventQueues()** first uses the function **vProcessIncomingMcps()** to accept the incoming data frame. Note that **vProcessIncomingMcps()** uses **vHandleMcpsDataInd()**, which calls **vProcessReceivedDataPacket()** in which you must define the processing to be done on the data.
   - When an event arrives in the hardware queue, **vProcessEventQueues()** calls the function **vProcessIncomingHwEvent()** to accept the incoming event. You must define the processing to be performed in this function.

Note: As it stands, the code is only designed to receive data. To transmit data from the device, you must modify the code - a transmission function is provided (see Section 4.6).

The above End Device set-up process is illustrated in Figure 2 below.

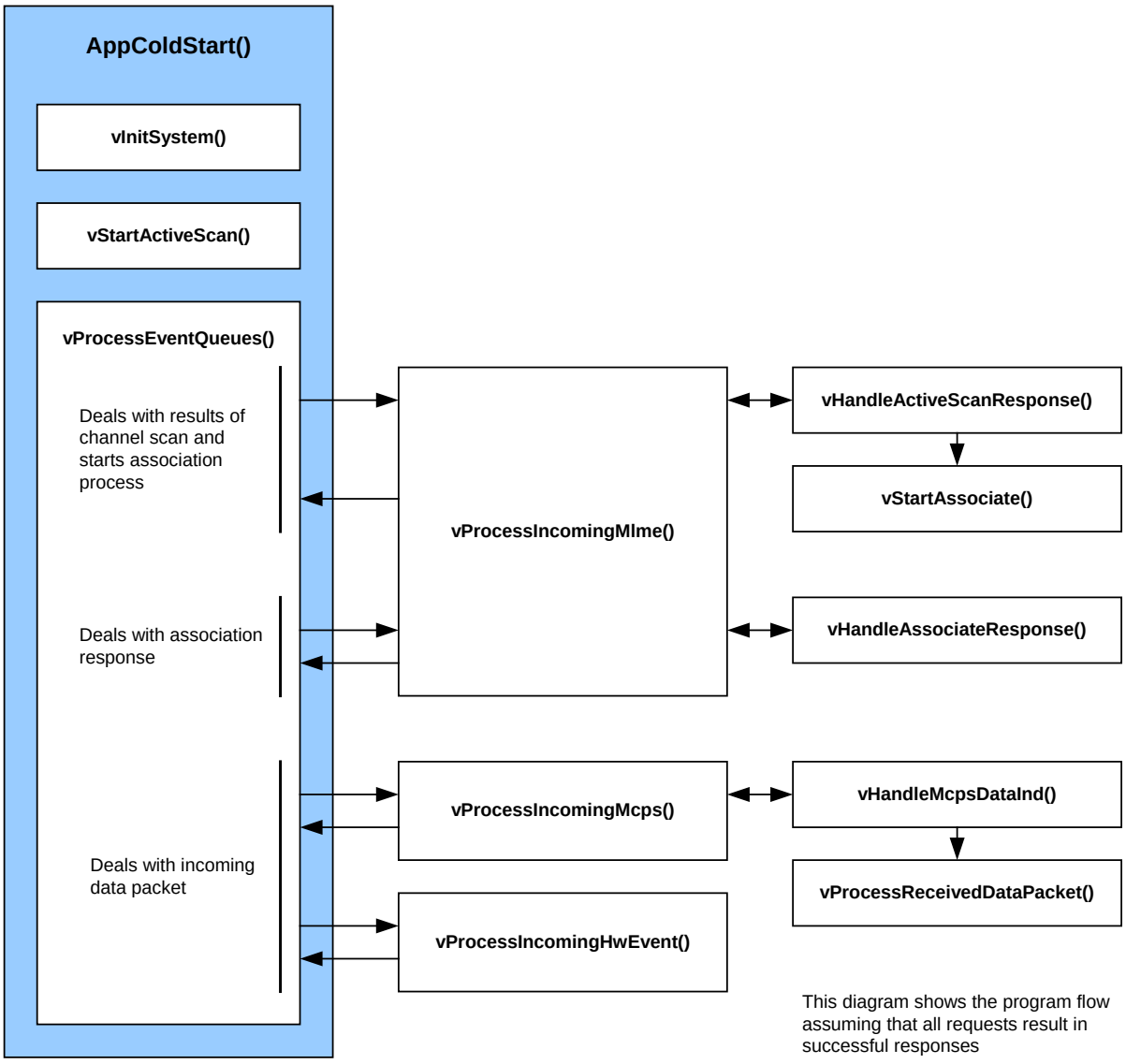

Figure 2: End Device Set-up Process

4 Adapting the Skeleton Code

- This section provides guidelines on how to modify the supplied skeleton code to achieve different requirements. The modifications covered are:
- How Do I Add End Devices to the Network?
- How Do I Program the Channel Scans?
- How Do I Define the Processing of Received Data Packets?
- How Do I Program Data Transmission?
- How Do I Program a Pre-defined PAN ID?
- How Do I Program Pre-defined Short Addresses?
- How Do I Add End Devices to the Network?
- How Do I Program the Channel Scans?
- How Do I Define the Processing of Received Data Packets?
- How Do I Program Data Transmission?

4.1 How Do I Program a Pre-defined PAN ID?

The PAN ID is pre-defined in the file **config.h**. In the skeleton code, it is set to 0xCAFE.

To use a different PAN ID, open **config.h** and change the hex number in the following line:

```
#define PAN_ID 0xCAFE
```

> **!** ***Caution:*** *The chosen PAN ID must not conflict with the PAN IDs of any other IEEE 802.15.4-based networks in the vicinity.*

4.2 How Do I Program Pre-defined Short Addresses?

The 16-bit short addresses of the PAN Co-ordinator and End Device are pre-defined in the file **config.h**. In the skeleton code, the short addresses are set to 0x0000 for the Co-ordinator and 0x0001 for the first End Device. The latter is a start address for the End Devices - if you have multiple End Devices, their short addresses will be automatically numbered from this value upwards in increments of 0x0001.

To use different short addresses, open **config.h** and change the hex numbers in the following lines:

```
#define COORDINATOR_ADR 0x0000
#define END_DEVICE_START_ADR 0x0001
```

> **Note:** It is usual to set 0x0000 as the short address of the PAN Co-ordinator.

4.3 How Do I Add End Devices to the Network?

The skeleton code is designed for a network consisting of at least two devices - a PAN Co-ordinator and an End Device. By default, the maximum number of End Devices defined in the code is 10 - this means that you can use up to ten End Devices without any modifications. However, you can use more End Devices by modifying the code as described below.

Note: When using multiple End Devices, their short addresses are automatically assigned starting with the address END_DEVICE_START_ADR defined in the **config.h** file (see Section 4.2)

Modifications to config.h

The file **config.h** contains a line defining the maximum number of End Devices supported by the application - in the supplied code, it is set to 10, as shown below:

```
#define MAX_END_DEVICES 10
```

To increase or decrease the maximum number of End Devices, open **config.h** and change this number.

Modifications to AN1174_154_EndD.c

The source file **AN1174_154_EndD.c** provides the code to be loaded into an End Device. If you have more than one End Device and they are of different types (e.g. one a temperature sensor, the other a humidity sensor), they are likely to need different source code. Therefore, when adding End Devices, you may need to devise specific code for the new devices.

Modifications to AN1174_154_Coord.c

To add End Devices to your network, you do not need to modify the file **AN1174_154_Coord.c**.

4.4 How Do I Program the Channel Scans?

The skeleton code involves two frequency channel scans:

- An Energy Detection Scan invoked by the PAN Co-ordinator during network set-up to find the most suitable channel for network operation.
- An Active Channel Scan invoked by the End Device during device association to find the operating channel of the PAN Co-ordinator.

It is not normally necessary to check all possible frequency channels. The 27 channels (numbered 0 to 26) of the IEEE 802.15.4 standard are distributed among the three frequency bands (868, 915 and 2400 MHz). Since a network is usually intended to work in only one of these bands, there is little point in scanning channels in the other two bands (NXP products operate in the 2400-MHz band; channels 11 to 26). In addition, you may be aware that another network in the locality already operates in one of the channels, so this channel should be excluded from the scan. Therefore, you can pre-define the channels that will be checked in these scans. You can also define the amount of time spent checking each channel in each of the scans. These definitions are made in the header file **config.h**, as described below.

Defining the Channels to be Scanned

The file **config.h** includes the following line:

```
#define SCAN_CHANNELS 0x07FFF800UL
```

SCAN_CHANNELS defines exactly which channels will be scanned. Each bit of the value (0x07FFF800 in this case) corresponds to a channel, where the least significant bit (LSB) corresponds to channel 0; see Figure 3.

- A bit value of 1 means 'scan'
- A bit value of 0 means 'do not scan'

To change the channels to be scanned, modify this hex value.

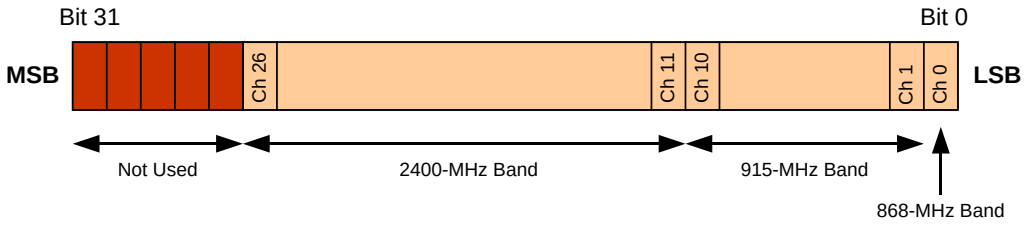

Figure 3: Channel Allocations in SCAN_CHANNELS

Note: SCAN_CHANNELS applies to both the Energy Detection Scan and the Active Channel Scan.

Caution: *Since the JN516x wireless microcontroller only operates in the 2400-MHz band, there is no point in configuring scans in channels of the lower bands.*

Defining the Channel Scan Durations

The file **config.h** includes the following two lines:

```
#define ACTIVE_SCAN_DURATION 3
#define ENERGY_SCAN_DURATION 3
```

Each of these parameters defines the time taken to check each channel in a scan:

- ACTIVE_SCAN_DURATION for an Active Channel Scan
- ENERGY_SCAN_DURATION for an Energy Detection Scan

These parameters take a positive integer value that determines the scan duration per channel, in milliseconds, according to the following formulae:

For an Active Channel Scan:

$$\text{Channel scan duration (ms)} = 15.36 \times (2^{\text{ACTIVE\_SCAN\_DURATION}} + 1)$$

For an Energy Detection Scan:

$$\text{Channel scan duration (ms)} = 15.36 \times (2^{\text{ENERGY\_SCAN\_DURATION}} + 1)$$

Thus, in each case, a value of 3 gives a channel scan duration of 138.24 ms. To change the channel scan durations, modify the above code values.

Note: The value of each of ACTIVE_SCAN_DURATION and ENERGY_SCAN_DURATION must be an integer in the range 0 to 14 (inclusive). Thus, the channel scan durations can be in the range 30.72 ms to 251.6736 s.

4.5 How Do I Define the Processing of Received Data Packets?

The IEEE 802.15.4 stack puts an incoming data packet into the MCPS queue on the destination device. The skeleton code for both the PAN Co-ordinator and End Device will retrieve the data packet from the queue but will not process the data in any way - you must define how you want to process the data. However, an empty function already exists in the code to accommodate your data processing code - **vProcessReceivedDataPacket()**. You must define the required processing for this function in the files **AN1174_154_Coord.c** and **AN1174_154_EndD.c**.

Note: The empty **vProcessReceivedDataPacket()** function appears in both **AN1174_154_Coord.c** and **AN1174_154_EndD.c**. However, the PAN Co-ordinator is likely to process received data packets in a different way from an End Device. Therefore, you are likely to define **vProcessReceivedDataPacket()** differently in the two source files.

4.6 How Do I Program Data Transmission?

In each of the source files **AN1174_154_Coord.c** and **AN1174_154_EndD.c**, a function for transmitting data is already defined - **vTransmitDataPacket()**. You simply need to add code to call this function as appropriate for your application.

5 Building and Downloading the Application

These applications can be built for the JN516x microcontrollers using BeyondStudio for NXP or makefiles.

Build the application as described in the appropriate section below, depending on whether you intend to use BeyondStudio or makefiles.

5.1 Using BeyondStudio

To build one of the applications and load it into a JN516x-based module, follow the instructions below (only one application can be present in Flash memory at a time):

1. Ensure that the project directory is located in

   <BeyondStudio for NXP installation root>\workspace

   where **<BeyondStudio for NXP Installation root>** is the path into which BeyondStudio for NXP was installed (by default, this is **C:\NXP\bstudio_nxp**). The **workspace** directory is automatically created when you start BeyondStudio for NXP.

2. Start BeyondStudio for NXP and import the relevant project as follows:

   a) In BeyondStudio, follow the menu path **File>Import** to display the Import dialogue box.

   b) In the dialogue box, expand **General**, select **Existing Projects into Workspace** and click **Next**.

   c) Enable **Select root directory** and browse to the **workspace** directory.

   d) In the **Projects** box, select the project to be imported and click **Finish**.

3. Build an application. To do this, ensure that the project is highlighted in the left panel of BeyondStudio and use the drop-down list associated with the hammer icon in the toolbar to select the relevant build configuration - once selected, the application will automatically build. Repeat this to build the other application.

   The binary files will be created in the relevant **Build** directory.

4. Load the resulting binary files into the boards. Do this using the integrated Flash programmer, as described in the *BeyondStudio for NXP Installation and User Guide (JN-UG-3098)*.

5.2 Using Makefiles

Each application has its own **Build** directory, which contains the makefiles for the application.

To build each application and load it into a JN516x-based board, follow the instructions below:

1. Ensure that the project directory is located in

   <BeyondStudio for NXP installation root>\workspace

   where **<BeyondStudio for NXP Installation root>** is the path into which BeyondStudio for NXP was installed (by default, this is **C:\NXP\bstudio_nxp**). The workspace directory is automatically created when you start BeyondStudio for NXP.

2. Start an MSYS shell by following the Windows Start menu path:

   All Programs > NXP > MSYS Shell

3. In the command window, navigate to the **Build** directory for the application to be built and follow the instructions below for your chip type:

   For JN5168:

   At the command prompt, enter:

   ```
   make clean all
   ```

   Note that for the JN5168, you can alternatively enter the above command from the top level of the project directory, which will build the binaries for both the applications.

   For JN5169:

   At the command prompt, enter:

   ```
   make JENNIC_CHIP=JN5169 clean all
   ```

   For JN5164:

   At the command prompt, enter:

   ```
   make JENNIC_CHIP=JN5164 clean all
   ```

   For JN5161:

   At the command prompt, enter:

   ```
   make JENNIC_CHIP=JN5161 clean all
   ```

   In all the above cases, the binary file will be created in the **Build** directory for the application, the resulting filename reflecting the name of the source file and the chip type (e.g. JN5168) for which the application has been built.

4. Load the resulting binary file from the **Build** directory into the boards. You can do this from the command line using the JN51xx Production Flash Programmer (described in the *JN51xx Production Flash Programmer User Guide (JN-UG-3099)*).

Revision History

Version	Notes
1.0	First release
1.1	Documentation references updated
1.2	Updated for BeyondStudio
1.3	Software updated for the JN5169 device
1.4	Document updated to incorporate full details of the application template

Important Notice

NXP Semiconductors

For the contact details of your local NXP office or distributor, refer to:

www.nxp.com