Application Note: JN-AN-1180

802.15.4 Home Sensor Demonstration for JN516x

This Application Note accompanies the source code (and associated files) of the Home Sensor Demonstration application for IEEE 802.15.4 networks that use NXP JN516x wireless microcontrollers. In this document, we introduce and describe this example application.

1 Application Overview

The 802.15.4 Home Sensor Demonstration application supplied in this Application Note is intended as an aid to understanding how an application can be built on top of the IEEE 802.15.4 stack on an NXP JN516x device. This application can be run on boards from the JN516x-EK001 Evaluation Kit.

The demonstration uses one board (fitted with an LCD panel and control buttons) as a controller node and a number of other boards as sensor nodes which measure temperature and humidity. The sensor boards periodically send sensor measurements to the controller node through a beacon mechanism. The controller node displays the received data on its LCD panel.

In this application, the controller node acts as the PAN Co-ordinator and the sensor nodes act as End Devices. Separate code is provided for the Co-ordinator and End Devices.

This Application Note:

- provides a high-level view of the application in terms of its hardware components and operation
- outlines the application design in terms of the software used, and the use of callbacks and interrupts
- describes how to build the application and download it to the hardware
- describes the application code, including the functions used
- outlines the optional IEEE 802.15.4 MAC-level security (disabled by default)

You will also need to refer to the source code that is zipped up with this Application Note (see Section 3.2).

All resources referenced in this document are available free-of-charge from the NXP Wireless Connectivity TechZone.

2 Compatibility

The software provided with this Application Note is intended to be used with the following NXP kits and SDK versions:

Product Type	Part Number	Version
Evaluation Kit	JN516x-EK001	-
SDK Libraries	JN-SW-4163	1281
BeyondStudio for NXP	JN-SW-4141	1217

3 Components

The hardware and software components required for the 802.15.4 Home Sensor Demonstration are indicated in the sub-sections below.

3.1 Hardware Components

The 802.15.4 Home Sensor Demonstration uses boards from the JN516x-EK001 Evaluation Kit, as follows:

- **Controller node x 1:** Comprises a DR1174 Carrier Board fitted with a DR1215 LCD Expansion featuring an LCD panel and four control buttons. This node acts as the IEEE 802.15.4 PAN Co-ordinator.
- **Sensor node x 3:** Each comprises a DR1174 Carrier Board fitted with a DR1175 Lighting/Sensor Expansion Board featuring a temperature sensor, humidity sensor, light sensor, one control button and two LEDs. These nodes act as IEEE 802.15.4 End Devices.

Note: The demonstration allows up to four sensor nodes to be used and uniquely labelled (as Hall, Bedroom, Lounge, Bathroom). Therefore, if an additional DR1175 Lighting/Sensor Expansion Board is available, a fourth sensor node can be included.

The demonstration allows the boards to emulate a home sensor and control system. The controller node allows each sensor node to be monitored, and alarms to be set for temperature and light levels on the sensor nodes. The controller node can be set to operate on a specific channel to avoid interference on busy frequencies, and the sensor nodes automatically scan for the controller node and synchronise with it.

3.2 Software Components

The Application Note package contains the source code for the demonstration application and pre-built binary files, as follows:

- **Co-ordinator (Controller Node)**
    - Source file **AN1180_154_HomeSensorCoord.c** is located in the folder **AN1180_154_HomeSensorCoord/Source**
    - Binary file **AN1180_154_HomeSensorCoord.bin** is located in the folder **AN1180_154_HomeSensorCoord/Build**
- **End Device (Sensor Node)**
    - Source file **AN1180_154_HomeSensorEndD.c** is located in the folder **AN1180_154_HomeSensorCoord/Source**
    - Binary file **AN1180_154_HomeSensorEndD.bin** is located in the folder **AN1180_154_HomeSensorCoord/Build**

You can load the binary files into the evaluation kit boards and immediately start to use the demonstration. To load the demonstration into the boards, refer to Section 4. Operational information is then provided in Section 5.

You can modify and re-build the applications according to your own requirements. You are advised to carry out this development work in BeyondStudio for NXP, which is an Eclipse-based IDE (Integrated Development Environment) provided in the package JN-SW-4141.

Eclipse project files are provided in the top level of the Application Note package and makefiles are provided in the **Build** folders of the respective applications. For details of how to build the applications, refer to Section 8.

4 Loading the Application

The application binary files (in the **Build** folders) must be loaded into the JN516x modules on the Carrier Boards of the relevant nodes, as follows:

- **AN1180_154_HomeSensorCoord.bin** must be loaded into the JN516x module of the Co-ordinator (controller node)
- **AN1180_154_HomeSensorEndD.bin** must be loaded into the JN516x module of each End Device (sensor node)

To load the files, you should use the Flash programmer that is integrated into BeyondStudio for NXP, as described in the *BeyondStudio for NXP Installation and User Guide (JN-UG-3098)*.

5 Operating Instructions

This section describes the buttons and screens used in the operation of the controller and sensor nodes.

5.1 Controller Node

The controller node (which acts as the PAN Co-ordinator) has several modes:

- Introductory splash screen, with channel selection
- Network display
- Individual node display
- Individual node control

The modes are accessed using the four buttons under the LCD panel on the controller node, with the hierarchy of screens shown below.

5.1.1 Button Conventions

The four buttons on the LCD Expansion Board of the controller node are used to navigate around a menu system. The buttons change their functions depending on the screen that is being displayed. On all screens, the function of a particular button/key is shown along the bottom of the LCD panel above the button/key position.

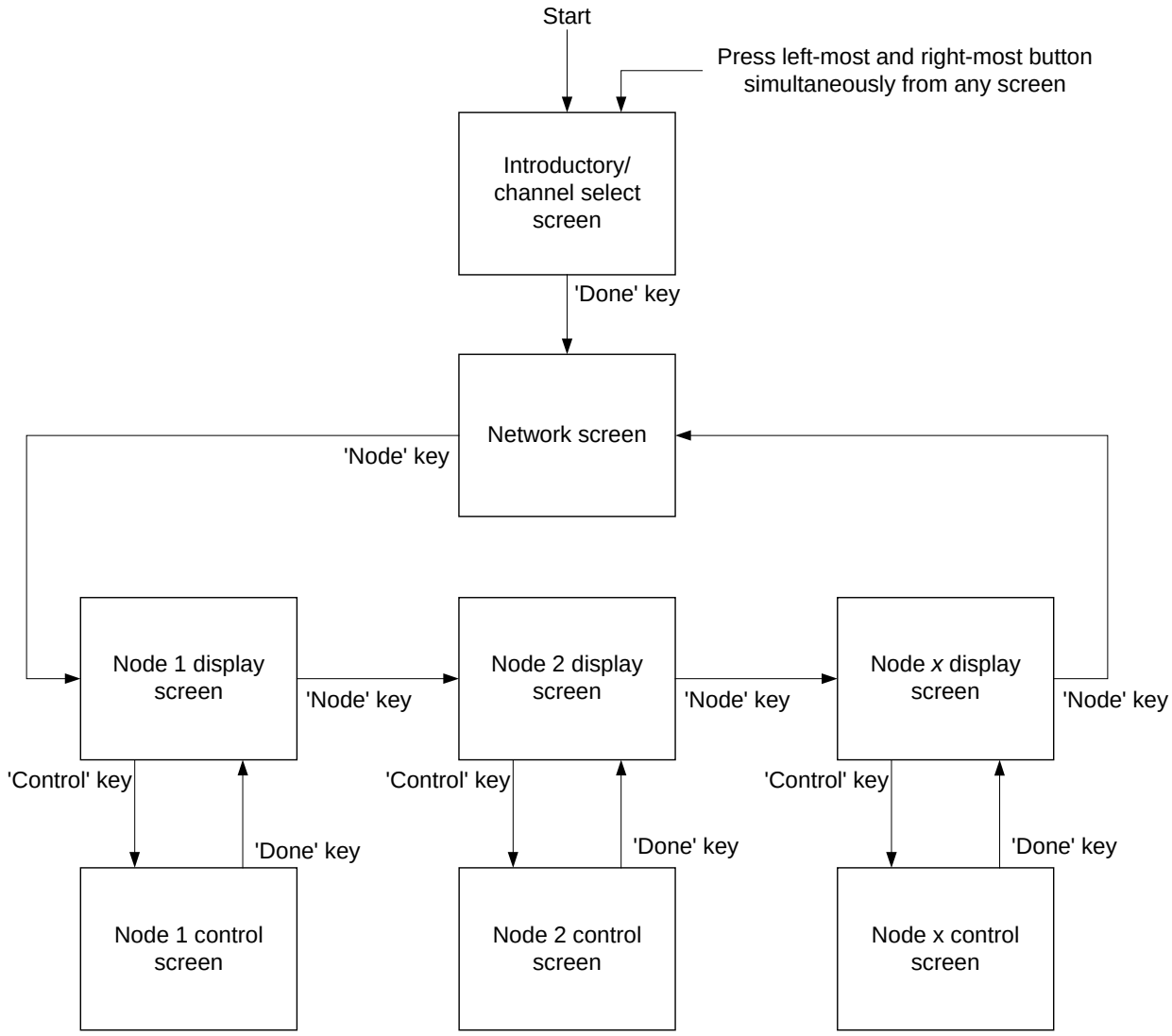


Figure 1: Controller Screen Flow Diagram

5.1.2 'Introductory' Screen

The 'Introductory' screen allows the operating channel to be set. Press '+' or '-' to adjust the channel number up or down. When the value reaches the top or bottom of the range (11 to 26), it wraps around. The initial value is 18.

When the desired channel has been selected, press 'Done' for the channel to be programmed into the hardware and to move to the Network screen.

The Introductory screen can be returned to at any time by pressing the left-most and right-most keys at the same time.

5.1.3 'Network' Screen

The 'Network' screen shows the current value and trend graph for one sensor type and for all sensor nodes simultaneously. It is possible to choose the sensor type to be displayed by pressing the 'Temp', 'Humidity' or 'Light' button - the currently selected sensor type is indicated by the corresponding button label being inverted.

In addition to the value and graph, any triggered alarms are displayed with the text 'High' or 'Low' in the space under the room name. If no alarms have been triggered, this space is used to display the link quality value for the relevant sensor node and the number of expected frames that have been missed. This information will not be displayed for the local node, as it is not applicable.

Note: Link quality gives a broad indication of the quality of the communication. The value is between 0 (for a very bad connection) and 255 (for an ideal connection). The value is affected by both the number of errors seen and the signal strength (which depends on the environment and the proximity of the devices).

To select a 'Node Display' screen, the 'Node' button should be pressed. If there are no nodes (i.e. 'Local node' has been set to 'off' on the controller node and no sensor nodes have yet associated), the 'Node' button has no effect.

5.1.4 'Node Display' Screen

The 'Node Display' screen shows the current value and trend graph for all sensor types simultaneously for one sensor node at a time. The format is the same as for the 'Network' screen, described above. The 'Node Display' screen also displays the link quality and number of frames that have been missed.

To select another sensor node, the 'Node' button should be pressed. When the final sensor node is displayed, pressing the 'Node' button again causes the 'Network' screen to appear.

Pressing the 'Control' button causes the 'Node Control' screen associated with the currently displayed sensor node to appear (see below).

Pressing 'On' or 'Off' controls the remote switch at the sensor node, as described in Section 5.2.2, unless the node is the controller node itself, in which case this option has no effect.

5.1.5 'Node Control' Screen

The 'Node Control' screen allows alarms and a remote switch to be set for a sensor node. The currently selected item is highlighted by the text for that item being inverted. To move from one item to the next, the 'Select' button should be pressed. When the last item has been selected, a subsequent press of the 'Select' button causes the first item to be selected again.

To change a numeric value, the '+' and '-' buttons can be used. When the value reaches the top or bottom of the range, it wraps around via an 'off' setting.

When all items are satisfactory, the 'Done' button should be pressed to return to the 'Node Display' screen associated with the sensor node.

5.2 Sensor Node

A sensor node uses the white LED cluster as well as the temperature/humidity and light sensors on the Lighting/Sensor Expansion Board.

The node can be in one of two modes: synchronising or operating.

5.2.1 Synchronising Mode

When a sensor node is switched on, it automatically enters synchronising mode. It will also return to this mode if it loses synchronisation with the controller node.

1. First the sensor node performs a channel scan. It repeatedly tries until it finds the channel on which the controller node is operating.
2. The node then synchronises with the controller node and associates with it. Again, this will be repeated until successful. During initial association, the sensor node is assigned a role by the controller node. The roles are assigned in the order in which each sensor node associates, in the order Hall, Bedroom, Lounge and Bathroom.

   The first sensor node to associate will be assigned ‘Hall’, the second ‘Bedroom’, etc. Therefore, if three sensor nodes are used in the demonstration, they will be assigned the labels ‘Hall’, ‘Bedroom’ and ‘Lounge’.
3. After a first successful association, the controller node remembers the role it assigned to a particular sensor node. Therefore, if that sensor node loses the communication link and has to re-associate, it will be assigned the same role again.
4. Once associated, the sensor node moves into operating mode (see next section).

5.2.2 Operating Mode

While in operating mode, the sensor node responds to every beacon from the controller node by sending back a frame containing sensor values. The controller node can then display this information.

The white LED cluster is controlled by the remote switch setting on the ‘Node Control’ screen on the controller node (see Section 5.1.5). The RGB LED is set to a steady state depending on the light level at the sensor node and the setting of the low light level alarm from the ‘Node Control’ screen on the controller node (see Section 5.1.5). The RGB LED is illuminated when the light level drops below the low alarm level and the alarm is set.

Note that each sensor is only read once per second in operating mode, and not at all if communication with the controller is lost.

6 Application Design

This section describes the architectural and certain operational aspects of the demonstration application.

6.1 Software Architecture

An application sits above the IEEE 802.15.4 stack which sits directly above the baseband hardware. The stack is provided with defined entry points to request 802.15.4 actions, and to initialise and register callbacks to the application.

A number of NXP Application Programming Interfaces (APIs) are used by the application. The 802.15.4 API sits at the top of the IEEE 802.15.4 stack. The Integrated Peripherals API and Board API sit logically to the side of the stack and are independent of it. This demonstration uses the optional Application Queue API, which also sits at this level.

This architecture is illustrated in Figure 2 below.

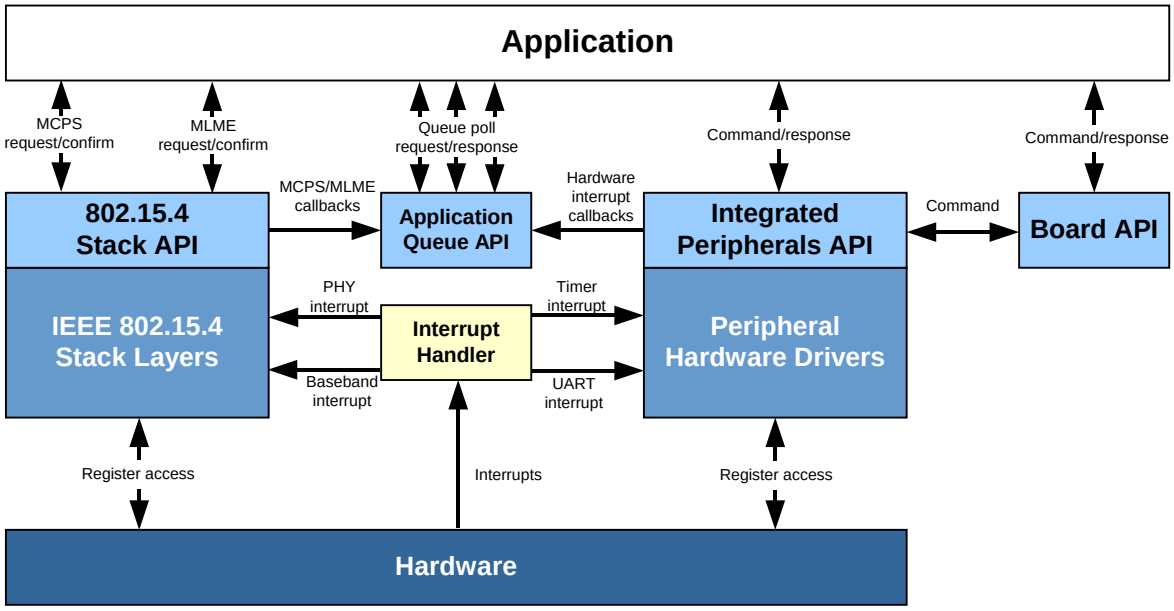


Figure 2: Software Architecture and API Usage

The APIs shown in the above diagram are detailed in NXP documentation as follows:

- The 802.15.4 Stack API is fully described in the *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*
- The Application Queue API is also described in the *IEEE 802.15.4 Stack User Guide (JN-UG-3024)*
- The Integrated Peripherals API is described in the *JN516x Integrated Peripherals API User Guide (JN-UG-3087)*
- The Board API is described in *LPRF Board API Reference Manual (JN-RM-2003)*

6.2 Context, Interrupts and Callbacks

Any call into the stack through an API entry point is performed in the application task context.

Many of the possible 802.15.4 requests cause the stack to initiate activities that will continue after the call has returned, such as a request to transmit a frame. In such cases, the stack will acquire processor time by responding to interrupts from the hardware. To avoid the need for a multi-tasking operating system, the stack will then work for as long as necessary in the interrupt context.

When information must be sent to the application (because of a previous request or due to an indication from the stack or hardware), the appropriate callback function is used.

All interrupts are generated by hardware. An interrupt handler in software decides whether to pass each interrupt to the 802.15.4 stack or to the peripheral hardware drivers, and these in turn either process the interrupt themselves or pass it up to the application via one of the registered callbacks.

7 Code Description

This section provides details of the code used in the demonstration application. This should help you understand the code sufficiently to adapt the application.

There is a build option to use 802.15.4-2006 MAC-level security for the data frames that are passed from End Devices to the Co-ordinator. The configuration and use of this security are detailed in the relevant places throughout this description.

7.1 Overview

The demonstration system consists of a Co-ordinator (the controller node) and several End Devices (the sensor nodes). The general structure of the code in each is the same, with an initialisation followed by a main loop. In the main loop, interrupts are used extensively to synchronise operation, which allows the device to put the CPU to sleep for long periods while nothing is happening. The Co-ordinator sends out regular beacons containing a beacon payload of 8 bytes. The first byte of the beacon payload contains a specific value so that the End Devices can use this to verify that the Co-ordinator is running the demonstration. As each End Device associates with the Co-ordinator, it is given a short address with which to identify itself. In addition, the buttons/keys are only checked 20 times per second - this avoids the need for any key de-bounce software algorithm, without giving a perceived operating delay.

As each End Device is switched on, it scans all channels and, after detecting beacons, checks that the Co-ordinator is the one that it is looking for. It then performs a synchronisation and association. Once association is complete, the End Device enters a regular loop of reading its sensors and sending out a frame containing the sensor data. As the beacons from the Co-ordinator contain a payload, the End Device receives an MLME indication from the stack whenever a beacon arrives. This is used to trigger the next read of the sensors.

Security can be used for the sensor data, with the End Device encrypting the frame and adding an authentication code to it. At the Co-ordinator, the frame is decrypted and the authentication is checked. All of this is performed by the MAC layer, but the application must configure it. Each End Device is assigned a different security key - the value is mostly pre-configured but one byte is an integer value in the range 0-3 based on the short address of the End Device. So the security is configured in two steps. The majority of the configuration is performed during initialisation, but some is performed as each End Device joins the network.

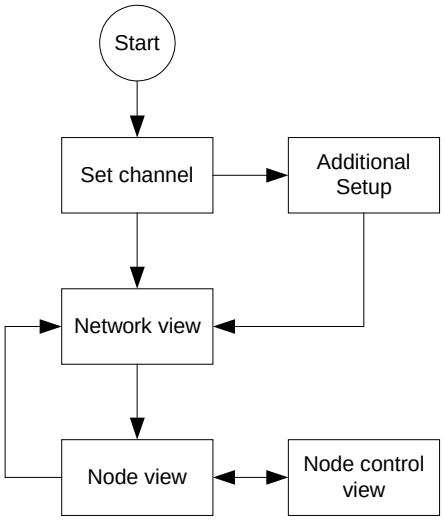
7.2 Co-ordinator

The Co-ordinator code (for the controller node) is contained in the file **AN1080_154_HomeSensorCoord.c** in the directory:

AN1080_154_HomeSensorCoord\Source

7.2.1 Overview

The general operation of the Co-ordinator is described in Section 5.1. It goes through a series of states that reflect which screen is being shown at any time, as indicated below.



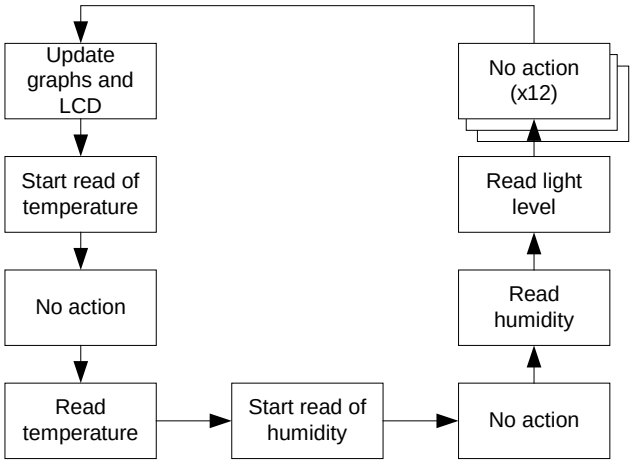
For each screen, there are associated functions in the code to create the screen, update the screen and handle any button-presses.

Upon being started, the first action of the application is the initialisation of the hardware, stack and application variables. Once this has completed, the first screen is shown and the main control loop is entered. This loop runs for the duration of the demonstration.

The main loop iterates 20 times per second, driven by events from a wake timer. Although the device is not sent to sleep between events, the application does put the CPU itself into doze mode whenever possible. The buttons are checked once per iteration of the loop, which avoids the need for any key de-bounce software algorithm, without adding any perceived operating delay. Any MLME or MCPS events are processed as they occur, handling association requests and received frames from End Devices.

Before first entering the Network view, the application sends an MLME request to the stack to start transmitting regular beacons. In addition, when showing the Network view, Node view or Node Control view, there is a simple state machine to co-ordinate the reading of sensors and the updates of the graphs and values displayed on the LCD panel. The screen is updated once per second unless user activity causes a change to the displayed information. The state machine runs from the same 20-Hz timer as the rest of the main loop, and the states are shown below.

Note: Although beacons are transmitted at a similar rate, the generation of beacons is performed by the stack and, once started, is completely independent of the application.



By reading the sensors some time after the read was initiated, the result is guaranteed to be ready so that no CPU effort is wasted in polling for the result.

7.2.2 Function Descriptions

The function descriptions in this section are intended to demonstrate how to create an IEEE 802.15.4 application using the NXP Application Programming Interfaces (APIs). As such, some functions are not mentioned as they are not directly relevant to this goal.

AppColdStart()

This function is the main entry point for the application, called after the ROM-resident boot loader has finished. It calls the initialisation function **vInitSystem()**, then uses a continuous loop to initialise the Co-ordinator and run the main loop. The main loop sets a timer, processes the state machine, processes any key presses and then waits for any MLME or MCPS confirmations/indications, or a hardware interrupt. The hardware interrupt is certain to occur, since the timer was set at the start of the loop. Once this interrupt fires, the loop re-starts.

AppWarmStart()

This function is required as the main entry point for the application after a warm start (i.e. the CPU has been powered down then restarted, with the RAM contents retained). This mode is not used in the demonstration application, so this function is included just to call **AppColdStart()** as a fail-safe mechanism.

InitSystem()

This function calls the Application Queue API initialisation function, which in turn calls the 802.15.4 Stack API initialisation function. The API is initialised without any callbacks. This function also calls the Integrated Peripherals API initialisation function and the relevant Board API initialisation functions.

The IEEE 802.15.4 PIB is written to, setting the PAN ID and short address for the device. These are determined at compile-time.

The wake timer used for the 20-Hz pulses is calibrated and then enabled here, but is not started.

If the security build option is enabled, most of the security configuration is performed here. The minimum security level required for data frames is 5, which means that any data frames received must be encrypted and authenticated with a MIC length of at least 4 bytes. The Co-ordinator supports four security keys, one for each possible End Device. These are set up in

the Key Descriptor table together with the Key ID Lookup table, which is used to find a key from an index value, and the Key Device Descriptor, which links the key to the correct entry in the Device Descriptor table. At this point, the Device Descriptor table is left blank, as the End Devices are not yet known.

InitCoord()

Data relating to the End Device sensor information is initialised here, as is the demonstrator configuration.

vSetTimer()

This function uses the Integrated Peripherals API to start the wake timer for 1600 cycles, a duration of approximately 1/20th of a second.

vProcessCurrentTimeBlock()

This function implements the state machine. It makes use of the Board API to access the sensors. The values returned are truncated, with temperature from 0 to 52 degrees centigrade, humidity from 0 to 104%, and light level as a value from 0 (dark) to 6 (light). The limits were chosen to allow easy conversion to a 0-to-13 scale for the graphs.

vProcessKeys()

This function relays any button-presses to the appropriate functions, depending on which screen is being shown. The buttons are all ‘soft’ – that is, their function is dependent on the screen.

vUpdateTimeBlock()

This function increments the state machine state, if the Co-ordinator is in the correct state. When some screens are shown, the state machine is effectively disabled.

vProcessInterrupts()

Once housekeeping tasks have completed, the Co-ordinator enters a continuous loop checking the MCPS, MLME and hardware queues for any incoming indications. Normally this would waste processor power, but the function causes the CPU to be first put into ‘doze’ mode. The result is that the CPU will remain unclocked until an interrupt occurs. The CPU will then wake up from where it left off. In this case, it will jump to the interrupt handler, process any interrupts and place any indications into the MCPS, MLME and hardware queues in the Application Queue API. Once interrupt processing has completed, this function is allowed to continue. It can check all of the queues and, once any processing has completed, loop back to start the whole process again.

Once a wake-up interrupt has been detected, the loop is exited and the function returns. This allows the main loop in **AppColdStart()** to continue.

vProcessIncomingData()

This function processes any MCPS data indications. Any other MCPS indication or confirmation is discarded, as the application does not expect any. Also, any received frame is discarded that does not have the correct payload length, or the pre-determined identifier as the first byte of the payload, or a short address within the expected range.

The short address of the received frame is used to determine which End Device it came from. This is then used as an index into an array containing the stored End Device data, which is read from the frame payload together with a control for an LED.

Note that if the security build option is enabled then a received frame which is not encrypted, or for which security information cannot be found, will be discarded by the MAC layer and it will not be passed to this function.

vProcessIncomingMlme()

Only **MLME.Associate** indications are expected in this application. When such an indication is received, the function checks whether the extended address of the requesting node is within the valid range. If this is the case, a short address is assigned. If the End Device has previously associated (for instance, if it associated and subsequently went out of radio range), it is given the same short address as before. If not, it is given the next available short address. A table of extended and short addresses is maintained to support this functionality.

Note that if the security build option is enabled, the Device Descriptor table entry is updated to include the short address, extended address and PAN ID of the End Device, and also to reset the frame counter. An association response is then created with the short address (if one has been assigned) and a suitable response code. If an End Device has been added to the system and the 'Network' screen is being displayed, it is updated to incorporate the new node. For any other screen type, this is not necessary.

bProcessForTimeout()

This function handles all Integrated Peripherals API indications, checking for a wake timer interrupt and returning TRUE if one has been found.

vProcessUpdateBlock()

This function updates the graphs for all nodes, scaling the data to fit, and then updates the display if the appropriate screen is being shown.

Key-Press Functions

vProcessSetChannelKeyPress()
vProcessNetworkKeyPress()
vProcessNodeKeyPress()
vProcessNodeControlKeyPress()
vProcessSetupKeyPress()

These functions all operate in much the same way, responding to button-presses to adjust values, set data or move to another screen.

vUpdateNetworkSensor()

This function is called to refresh the 'Network' screen, which shows the same sensor type from all nodes at once.

LCD Screen Functions

vBuildSetChannelScreen()
vBuildNetworkScreen()
vBuildNodeScreen()
vBuildNodeControlScreen()
vBuildSetupScreen()

These functions all demonstrate the creation of a fresh screen on the LCD panel using the Board API. Note that the font definition requires some odd characters to be used. For instance, '\' is used for a '+' and ']' is used for '-'. Normally, a call to **vLcdRefreshAll()** would be used to put the new screen onto the LCD panel, but in all cases an associated update function is called instead, which then calls **vLcdRefreshAll()**.

Extensive use is made of **vLcdWriteText()**. **vBuildSetChannelScreen()** also uses **svLcdWriteBitmap()**, with the bitmap that is defined in **NxpLogo.c**.

vUpdateSetChannelScreen()
vUpdateNetworkScreen()
vLcdUpdateElement()
vUpdateNodeScreen()
vUpdateNodeControlScreen()
vUpdateSetupScreen()

These functions all update an existing LCD screen while retaining what was there before. Note that space characters are used to delete unwanted text. Since the font is proportional, each space is only 3 pixels wide, while most characters are 5 pixels wide. It is therefore necessary to use additional space characters to successfully erase a series of text characters.

vDrawGraph()

This function takes an array of 32 values, in the range 0 to 13, and makes a bitmap graph from them. This is an alternative to using a pre-defined bitmap and allows simple graphics to be created without the complication of a line-drawing algorithm.

An array is created for the two rows of data used by each graph, and a constant array is defined containing the data values that map to each integer value in the range 0 to 13. The array is then filled with the appropriate data for each item in the series of values.

A structure defines the bitmap in terms of the array, as being 33 columns wide and 2 character rows high. Finally, a call to the Board API writes the bitmap to the LCD shadow memory. A call to **vLcdRefreshAll()** must be made after calling this function for the bitmap to appear on the LCD panel.

vStartBeacon()

This function generates an MLME request to start regular beacons. As this MLME request returns a confirmation, this is checked for any problems and an error is displayed if a problem occurs.

vUpdateBeaconPayload()

For debugging, the beacon payload is updated with some status information once per second. This requires two **MLME-Set.Request** calls, one for the payload contents, which are passed as an array of bytes, and one for the beacon payload length. In both cases, any confirmation is ignored, as the values provided cannot cause a failure.

vDisplayError()

This function displays an error message and then loops indefinitely. It is intended for fatal errors such as a failure to start beacons.

Miscellaneous Functions

vStringCopy()
vValToDec()
vAdjustAlarm()

These functions do not make use of the stack or API at all, so are not described here.

Note: The remaining functions do not add clarity to the use of the stack or API, so are not described here.

7.3 End Device

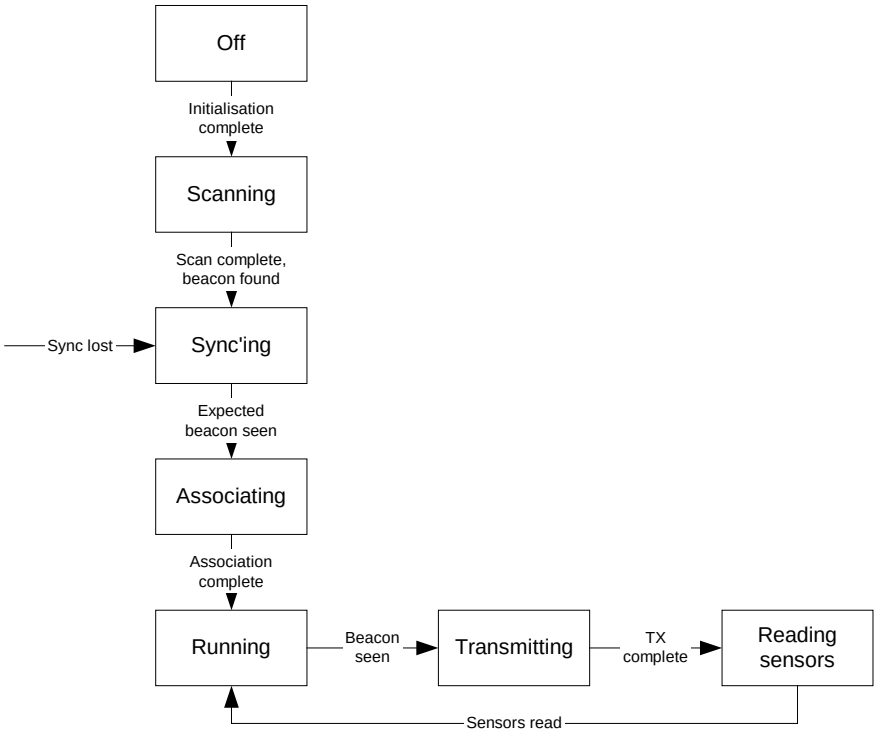
The End Device code (for the sensor nodes) is contained in the file **AN1080_154_HomeSensorEndD.c** in the directory:

AN1080_154_HomeSensorEndD\Source

7.3.1 Overview

Like the Co-ordinator code, the End Device code uses a main loop that simply processes interrupts and indications, using these to step through a state machine to find and associate with the Co-ordinator and then send sensor data to it. Unlike the Co-ordinator, the End Device does not use a timer and there is no button processing.

The state machine is as follows:



After initialisation, which starts the scan for the first time, all subsequent movements between states occur as the results of interrupts. The MLME processing function uses the present state and the received indication or confirmation to decide what to do next.

Note that the Co-ordinator sends beacons with a beacon payload, so there is always an MLME indication when they arrive. This is used to time the queuing of outgoing frames to ensure that the End Device application only tries to send one frame for every eight beacons (although in practice, this is only necessary for reducing power consumption). Any additionally transmitted frames would be wasted as the Co-ordinator only updates the LCD panel once per second, whereas the beacons are transmitted by the Co-ordinator eight times per second.

7.3.2 Function Descriptions

The function descriptions in this section are intended to assist in demonstrating how to create an IEEE 802.15.4 application using the NXP APIs. As such, some functions are not mentioned as they are not directly relevant to this goal.

AppColdStart()

This is the main entry point for the application, called after the bootloader has finished. It calls the initialisation function **vInitSystem()**, then uses a continuous loop to initialise the End Device, start a scan and then sit in the interrupt handler until a reset condition is detected.

AppWarmStart()

This function is required as the main entry point for the application after a warm start (i.e. the CPU has been powered down then restarted, with the RAM contents retained). This mode is not used in the demonstration application, so this function is included just to call **AppColdStart()** as a fail-safe mechanism.

vInitSystem()

This function calls the Application Queue API initialisation function, which in turn calls the 802.15.4 Stack API initialisation function. The API is initialised without any callbacks. This function also calls the Integrated Peripherals API initialisation function and the relevant Board API initialisation functions.

The IEEE 802.15.4 PIB is written to, setting the PAN ID for the device, which is determined at compile-time.

If the security build option is enabled, most of the security configuration is performed here. The minimum security level required for data frames is 5, which means that any data frames received must be encrypted and authenticated with a MIC length of at least 4 bytes. The End Device supports just one security key. This is set up in the Key Descriptor table together with the Key ID Lookup table, which is used to find a key from an index value, and the Key Device Descriptor, which links the key to the entry in the Device Descriptor table. Although there is only one entry in each of these tables, they must all be populated so that the configuration values can be linked together. At this point, the Device Descriptor table is left blank, as the Co-ordinator is not yet known.

vInitEndpoint()

This function sets parameters used by the End Device.

vStartScan(), vStartSync(), vStartAssociate()

These functions demonstrate how to send a request to the stack to start an active scan, synchronisation or association, respectively.

Starting a scan or an association can return an error in the confirmation, so this is checked for. The desired result is a deferred confirmation, meaning that the confirmation will arrive via the upward queues.

vProcessInterrupts()

This is a continuous loop checking the MCPS, MLME and hardware queues for any incoming indications. Normally this would waste processor power, but the function causes the CPU to be first put into ‘doze’ mode. The result is that the CPU will remain unclocked until an interrupt occurs. The CPU will then wake up from where it left off. In this case, it will jump to the interrupt handler, process any interrupts and place any indications into the

MCPS, MLME and hardware queues. Once interrupt processing has completed, this function is allowed to continue. It can check all of the queues and, once any processing has completed, loop back to start the whole process again.

The only interrupt expected from the Integrated Peripherals API is when a button is pressed. To minimise code, the cause of an Integrated Peripherals API interrupt is never checked. The buttons are examined whenever an Integrated Peripherals API interrupt occurs and if the reset combination is detected then the function will exit. There is no need to perform any key de-bounce due to the nature of the functions assigned to the keys.

vProcessIncomingMcps()

The only MCPS indication or confirmation that is processed is a deferred confirmation of a previous transmission attempt. Whether it succeeded or failed is not important – the MAC software will have attempted multiple retries automatically – so this function is just used to trigger the reading of the sensors ready for the next beacon.

vProcessIncomingMlme()

Deferred confirmations of scan and associations are processed here, as are beacon notification indications.

If in the scanning state and a scan confirmation arrives, there are two possibilities:

- The desired Co-ordinator was detected (identified by the PAN ID and short address which, in this application, are pre-determined). The channel is set to that used by the Co-ordinator and synchronisation is started.
- The desired Co-ordinator was not detected. Try scanning again.

If in the associating state and an association confirmation arrives, there are also two possibilities:

- The association was successful, so start using the supplied short address and enter the ‘running’ state. If the security build option is enabled, the Device Descriptor table entry is updated to include the short address, extended address and PAN ID of the Co-ordinator, and also to reset the frame counter.
- The association was not successful. Try associating again.

If a beacon notification indication is detected, there are two possibilities depending on the state:

- If synchronising, this indicates that synchronisation has occurred, so the device can start to associate.
- If ‘running’, a beacon has arrived, so a frame should be sent back with the sensor data.

vProcessRead()

This function reads the sensors via the Board API. The approach used is not the most efficient, as the processor will spend a period of time polling the humidity and temperature sensor while waiting for a result to be generated. An alternative approach would have been to enable an interrupt when DIO8 goes low, which is how the sensor indicates that a result is ready to be read, and to use this to start a read of the data via the **vProcessInterrupts()** function. The light sensor is better in this respect, as it performs continuous conversions, so the processor can read a value from it immediately.

vProcessTxBlock()

This function creates an **MCPS-Data.request**, using the network byte-order functions to convert 16-bit values into arrays of bytes. The short address provided by the Co-ordinator is used for the source address, and pre-determined values are used for the Co-ordinator short address and PAN ID.

If the security build option is enabled, the security data to be used for the frame transmission is specified - the security level to use, the Key ID mode and the Key ID. The last two are used by the MAC layer to look up the key by matching the values in the Key ID Lookup table.

This function does not check the immediate confirmation from the call into the 802.15.4 Stack API, as there is no recovery mechanism implemented in the case of failure. Different applications may want to check the confirmation if they have an action to perform in situations where an **MCPS-Data.request** is likely to fail under normal operation.

vProcessRxBeacon()

This function checks that the received beacon contained the correct beacon payload, identifying it as from the Co-ordinator of the demonstration application. The function retrieves the contents of the payload (the LED control value and light alarm level) for this node. All accesses to the payload are byte accesses, so there is no need to be concerned about byte ordering. A call to **vProcessTxBlock()** is made at the end of this function in order to queue a frame for transmission back to the Co-ordinator.

vDisplayError(), vDisplayHex(), vDebug()

These functions are optionally used to display error messages via serial port 0, using the Integrated Peripherals API. To compile in this feature, UART0_DEBUG must be defined in the makefile **HomeSensorEndDevice.mk**.

8 Building the Application

This section describes how to re-build the demonstration application, which you may want to do if you have made your own modifications to the source code.

8.1 Pre-requisites

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

In order to build the application, this Application Note (JN-AN-1180) must have been unzipped into the directory:

<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.

All files should then be located in the directory:

…\workspace\JN-AN-1180-JN516x-802-15-4-Home-Sensor-Demo

The files that are specific to the two device types (Co-ordinator and End Device) are contained in two separate sub-directories:

- **AN1080_154_HomeSensorCoord**
- **AN1080_154_HomeSensorEndD**

each having **Source** and **Build** sub-directories.

8.2 Build Instructions

The demonstration software can be built for the NXP JN5169, JN5168 and JN5164 devices. By default, it is built for the JN5168 device.

You will need to build the applications for the different device types (Co-ordinator, End Device) separately: **AN1080_154_HomeSensorCoord.c** and **AN1080_154_HomeSensorEndD.c**.

The applications can be built with or without security enabled. It is important to ensure that the Co-ordinator and the End Devices all use the same build option.

The applications can be built from the command line using the makefiles or from BeyondStudio for NXP – makefiles and Eclipse project files are supplied.

- To build using makefiles, refer to Section 8.2.1.
- To build using BeyondStudio for NXP, refer to Section 8.2.2.

8.2.1 Using Makefiles

This section describes how to use the supplied makefiles to build the demonstration application. The application for each node type (Co-ordinator, End Device) has its own **Build** directory, which contains the makefiles for the application.

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

1. Ensure that the project directory is located in

   <BeyondStudio for NXP installation root>\workspace

2. Start an MSYS shell by following the Windows Start menu path:
   All Programs > NXP > MSYS Shell

3. 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, to build without security enabled, enter:

   ```
   make clean all
   ```

   Alternatively, to build with security enabled, enter:

   ```
   make USE_SECURITY=1 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, to build without security enabled, enter:

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

   Alternatively, to build with security enabled, enter:

   ```
   make USE_SECURITY=1 JENNIC_CHIP=JN5169 clean all
   ```

For JN5164:

At the command prompt, to build without security enabled, enter:

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

Alternatively, to build with security enabled, enter:

```
make USE_SECURITY=1 JENNIC_CHIP=JN5164 clean all
```

In all the above cases, the binary file will be created in the **Build** directory, the resulting filename indicating the chip type (**5169**, **5168** or **5164**) for which the application was built. If built with security enabled, the filename will also include the postfix "_**SECURE**".

4. Load the resulting binary file into the board. 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)*).

8.2.2 Using BeyondStudio for NXP

This section describes how to use BeyondStudio for NXP to build the demonstration application.

To build the application and load it into JN516x boards, follow the instructions below:

1. Ensure that the project directory is located in

<BeyondStudio for NXP installation root>\workspace

2. Start the 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 applications.

The binary files will be created in the relevant **Build** directories for the applications.

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

Revision History

Version	Notes
1.0	First release
1.1	Software updated to include IEEE 802.15.4 MAC security and document also updated/re-worked
1.2	Updated for BeyondStudio
1.3	Number of sensor nodes in demonstration clarified
1.4	Updated to include the JN5169 device

Important Notice

NXP Semiconductors

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

www.nxp.com