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 the NXP JN5168 wireless microcontroller. Here, we introduce this example application and direct you to the relevant documentation that describes the application more fully.

1 Application Overview

The 802.15.4 Home Sensor Demonstration application is intended as an aid to understanding how an application can be built on top of the IEEE 802.15.4 stack on a JN51xx device and how to use the boards of an NXP evaluation kit fitted with these devices.

The demonstration uses a carrier board with an LCD Expansion board fitted (controller node), and a number of carrier boards with sensor expansion boards (sensor nodes) from the evaluation kit. The sensor nodes measure temperature and humidity, and periodically send these measurements to the controller node through a beacon mechanism. The controller node displays the received data on its built-in 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

You will also need to refer to the source code that is zipped up with this Application Note: **AN1180_154_HomeSensorCoord.c** for the Co-ordinator (controller node) and **AN1180_154_HomeSensorEndD.c** for the End Devices (sensor node).

2 Compatibility

The software provided with this Application Note has been tested with the following NXP kits and SDK versions:

Product Type	Part Number	Version	Supported Chips	Supported Protocols
Evaluation Kit	JN516x-EK001	-	JN5168, JN5164	802.15.4
SDK Libraries	JN-SW-4065 JN-SW-4063	-	JN5168	802.15.4
SDK Toolchain	JN-SW-4041	v1.1	JN5148, JN5168	-

3 Operational Features

The Home Sensor Demonstration application is intended as an aid to understanding how an application can be built on top of the IEEE 802.15.4 stack, and how to use the sensors and LCD panel of an NXP evaluation kit.

3.1 Hardware Components

The 802.15.4 Home Sensor Demonstration should be prepared from the JN516x-EK001 Evaluation Kit components to include:

- One **controller node** with an LCD panel and four control buttons, comprising a Carrier Board and LCD Expansion Board
- Three **sensor nodes**, each with a temperature sensor, humidity sensor and light sensor, one control button and two LEDs, comprising a Carrier Board and Lighting/Sensor Expansion Board

The 802.15.4 Home Sensor Demonstration allows the boards to emulate a home sensor and control system. This software is provided in the Application Note JN-AN-1050, which is available from www.nxp.com/jennic/support. Refer to Section 5 for details of how to build this application and download it to the evaluation kit boards.

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

Note: In this IEEE 802.15.4 application, the controller node acts as the PAN Co-ordinator and the sensor nodes are End Devices.

3.2 Operating Instructions

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

3.2.1 Controller Node

The central controller (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 by pressing buttons on the controller, with the hierarchy of screens shown below.

3.2.1.1 Button Conventions

The switches on the controller node are used to navigate around a menu system, and change their function depending on which screen is being shown. On all screens, the function of a particular switch is displayed on the bottom of the LCD panel above the switch position.

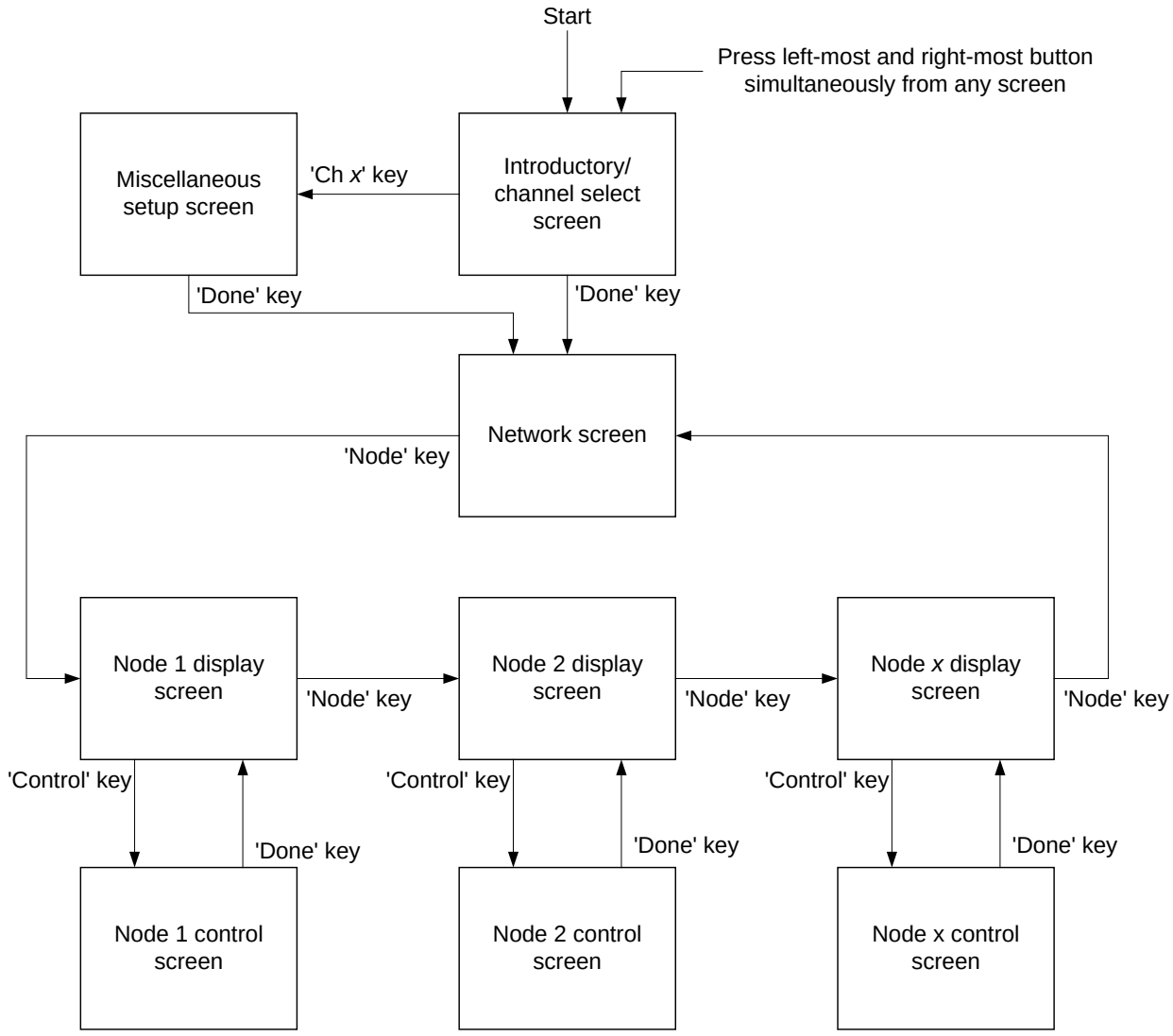


Figure 1: Controller Screen Flow Diagram

3.2.1.2 '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. Alternatively, press 'Ch x' to go to the Miscellaneous Settings screen.

The introductory screen can be returned to at any time by pressing the leftmost and rightmost keys at the same time.

3.2.1.3 'Miscellaneous Settings' Screen

The 'Miscellaneous Settings' screen is for non-standard configuration of the system. The options are as follows:

Name	Possible values	Default
Local node	off, on	off
Four nodes	off, on	on

The currently selected item is highlighted by the text for that item being inverted. To move between the two options, the 'Select' button should be pressed. To alter a value for the selected option, the '+' and '-' buttons can be used.

It is possible to associate up to four sensor nodes with the controller, and all displayed nodes will be sensor nodes.

Note: 'Local node' allows the controller node to be counted as a sensor node. However, this is not used in this demo.

In addition to the above options, this screen shows the version numbers for the 802.15.4 Stack API and Integrated Peripherals API.

After all settings have been completed, press 'Done' to move to the Network screen.

3.2.1.4 '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 display 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 that 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.

3.2.1.5 '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 3.2.2.2, unless the node is the central controller itself, in which case this option has no effect.

3.2.1.6 '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 alter 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.

3.2.2 Sensor Node

A sensor node comprises a Carrier Board and Lighting/Sensor Expansion Board. It uses the white LED cluster on the expansion board, and temperature/humidity and light sensors.

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

3.2.2.1 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.

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

   The first sensor node to associate will be assigned 'Hall', the second 'Bedroom', and so on.
3. After a first successful association, the controller 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.

3.2.2.2 Operating Mode

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

The White LED cluster is controlled by the remote switch setting on the 'Node Control' screen on the controller (see Section 3.2.1.6). 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 (see Section 3.2.1.6). 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.

4 Application Design

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

4.1 Software Architecture

An application sits above the 802.15.4 stack, which in turn 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.

The Integrated Peripherals API and Board API sit logically to the side of the 802.15.4 stack and are independent of it.

This is illustrated in the diagrams in the next section.

4.2 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. It must be remembered that the callback is still in the interrupt context and any activity performed by the application within the callback must be kept as short as possible.

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.

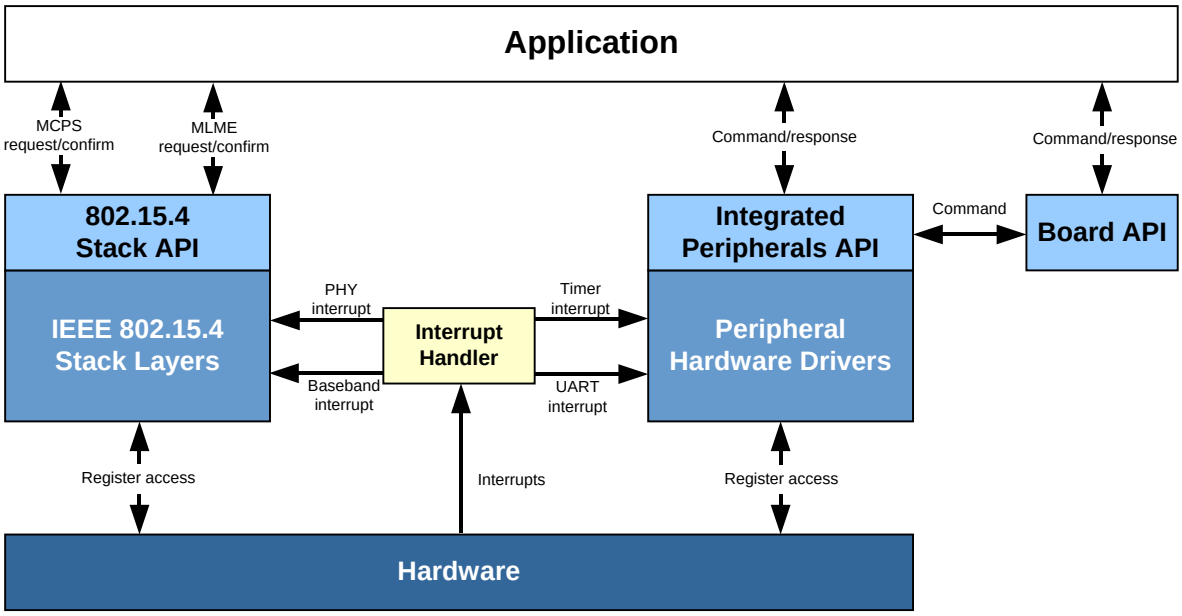


Figure 2: API Usage

The Integrated Peripherals API is described in the *JN516x Integrated Peripherals API User Guide (JN-UG-3087)*.

The 802.15.4 Stack API is fully described in the *802.15.4 Stack API Reference Manual (JN-RM-2002)*.

The Board API is described in *LPRF Board API Reference Manual (JN-RM-2003)*.

Alternatively, the optional Application Queue API can be used to handle interrupts for the application. This is illustrated in the diagram below. The Application Queue API is described in the *Application Queue API Reference Manual (JN-RM-2025)*.

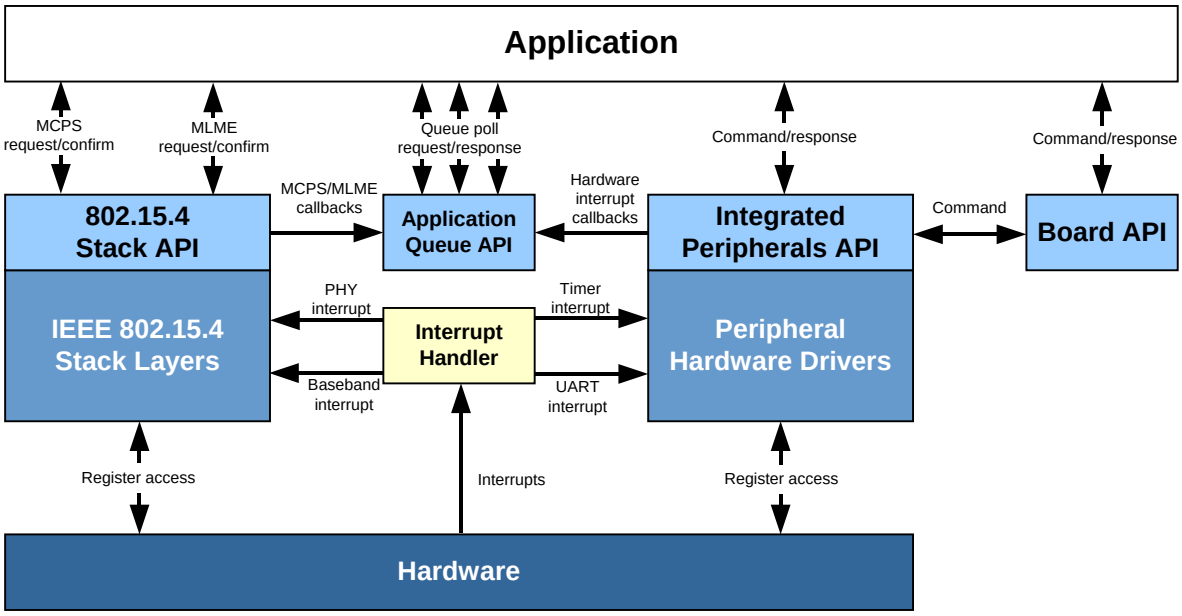


Figure 3: API Usage including Application Queue API

5 Unpacking, Building and Loading the Application

This section describes how to unpack, build and load the demonstration application.

It is assumed that you have installed one of the NXP JN516x SDKs on your PC – that is, JN-SW-4041 and any one of: JN-SW-4065, JN-SW-4064, JN-SW-4062, JN-SW-4060.

The above installers are available from www.nxp.com/jennic/support. When installing an SDK, follow the instructions provided in the *SDK Installation and User Guide (JN-UG-3064)*, also available from the above web address.

5.1 Unpacking the Application Note

In order to build the supplied software, first unzip this Application Note (JN-AN-1180) into

<JN516x_SDK_ROOT>\Application

where **<JN516x_SDK_ROOT>** is the path into which the JN516x SDK was installed (by default, this is **C:\Jennic**). The **Application** directory is automatically created when you install the SDK.

All files are then located in the directory

SDK\Application\JN-AN-1080-802-15-4-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.

5.2 Building and Loading the Application

This section describes how to build the demonstration application and load the resulting binary files into the JN516x-EK001 Evaluation Kit boards. 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 demonstration software can be built for the NXP JN5168 and JN5164 wireless microcontrollers.

The binary files produced in the build process are output to directories which depend on the build method, as indicated in the sub-sections that follow.

- To build using makefiles, refer to Section 5.2.1.
- To build using the Eclipse IDE, refer to Section 5.2.2.

5.2.1 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

   <JN516x_SDK_ROOT>\Application

   where **<JN516x_SDK_ROOT>** is the path into which the JN516x SDK was installed.

2. 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 JN5164:

   At the command prompt, enter:

   ```
   make 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 (**5168** or **5164**) for which the application was built.

3. Load the resulting binary file into the board. To do this, use the NXP JN51xx Flash Programmer, described in the *JN51xx Flash Programmer User Guide (JN-UG-3007)*.

5.2.2 Using Eclipse

This section describes how to use the Eclipse IDE 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

   <JN516x_SDK_ROOT>\Application

   where **<JN516x_SDK_ROOT>** is the path into which the JN516x SDK was installed.

2. Start the Eclipse platform and import the relevant project files (**.project** and **.cproject**) as follows:

   a) In Eclipse, 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**, browse to the NXP **Application** directory and click **OK**.

   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 Eclipse and use the drop-down list associated with the hammer icon  in the Eclipse

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** directories for the applications.

4. Load the resulting binary files into the boards. You can do this using the NXP JN51xx Flash Programmer, which can be launched from within Eclipse or used directly (and is described in the *JN51xx Flash Programmer User Guide (JN-UG-3007)*).

6 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.

6.1 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 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 any 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.

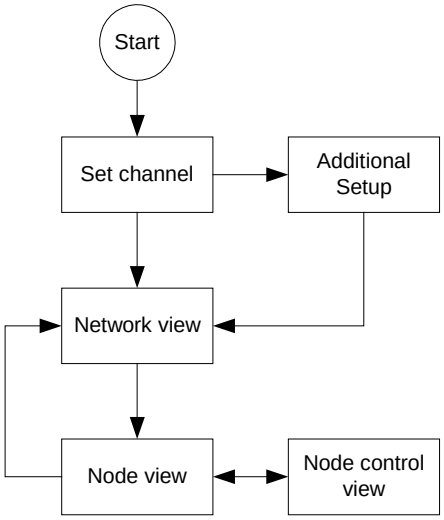
6.2 Co-ordinator

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

AN1080_154_HomeSensorCoord\Source

6.2.1 Overview

The general operation of the Co-ordinator is described in Section 3.2.1. It goes through a series of states that reflect which screen is being shown at any time, as indicated below.



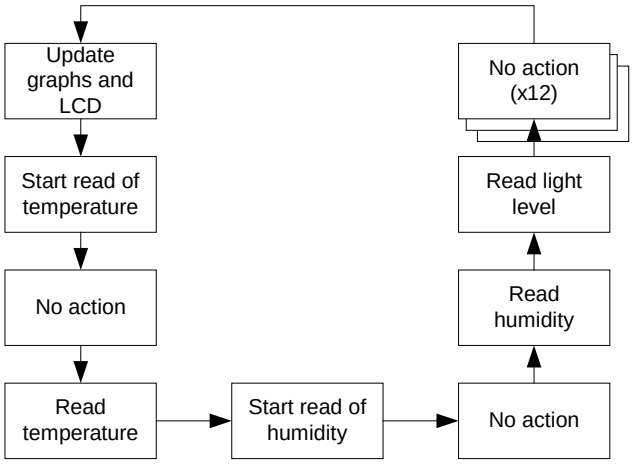
For each screen, there are associated functions in the code to create the screen, update the screen and handle any button presses.

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-up 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.

6.2.2 Function Descriptions

The function descriptions in this section are intended to demonstrate how to create an 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. The API is initialised without any callbacks. It also calls the Integrated Peripherals API initialisation function and the board initialisation functions (LCD, light sensor, humidity and temperature sensor).

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

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-up timer for 1600 cycles, 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.

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 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 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.

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 shown, 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-up 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.

Keypresses

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 screens

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

These functions all demonstrate the creation of a fresh screen on the LCD, 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 display onto the LCD, but in all cases an associated update function is called instead, which then calls **vLcdRefreshAll()**.

Extensive use is made of **vLcdWriteText()**, and **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 display 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 values required for each 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.

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

Miscellaneous

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.

6.3 End Device

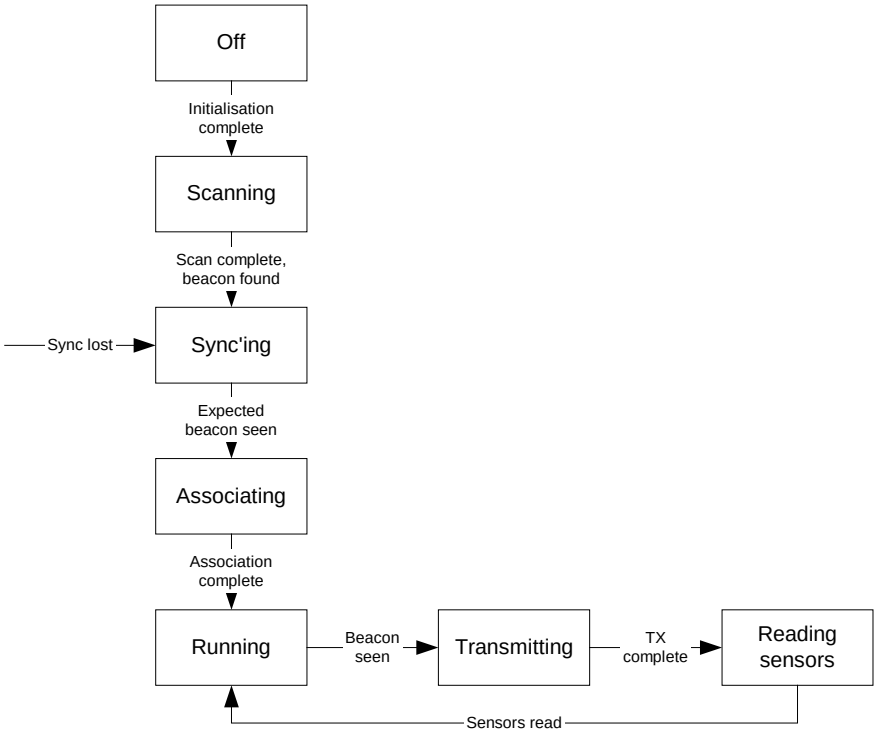
The End Device code (for the sensor nodes) is contained in the file **AN1080_154_HomeSensorEndD.c** in the directory:

AN1080_154_HomeSensorEndD\Source

6.3.1 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 the buttons are set to produce interrupts when they are pressed.

The state machine is as follows:



After initialisation, which starts the scan for the first time, all subsequent movements between states occur as 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 display once per second, whereas the beacons are transmitted by the Co-ordinator eight times per second.

6.3.2 Function Descriptions

The function descriptions in this section are intended to assist in demonstrating how to build an 802.15.4 application using the stack and 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 boot loader 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. The API is initialised without any callbacks. It also calls the Integrated Peripherals API initialisation function and the board initialisation functions (LCD, light sensor, humidity and temperature sensor).

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

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 both 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 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 confirmation of scan and associate 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 coordinator 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
- The association was not successful. Try associating again.

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

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

vProcessKeyPress()

This function uses the Integrated Peripherals API to determine the state of the buttons on the module. It then uses this result to either turn the remote switch on or off (value is passed to the Co-ordinator to control an LED) or to indicate a reset condition when both buttons are held down.

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 DIO 8 goes low, which is how the sensor indicates that a result is ready to 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.

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 level 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**.

Revision History

Version	Notes
1.0	First release

Important Notice

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

All trademarks are the property of their respective owners.

NXP Laboratories UK Ltd
(Formerly Jennic Ltd)
Furnival Street
Sheffield
S1 4QT
United Kingdom

Tel: +44 (0)114 281 2655
Fax: +44 (0)114 281 2951

www.nxp.com/jennic