Application Note: JN-AN-1236

ZigBee 3.0 Migration Guidelines for JN517x

This Application Note provides guidance on migrating ZigBee PRO applications, initially designed for the NXP JN516x wireless microcontrollers, to the latest stack supplied in the ZigBee 3.0 Software Developer's Kit (SDK) for the JN517x wireless microcontrollers. In particular, the ZigBee 3.0 SDK does not require the NXP proprietary operating system, JenOS, and therefore guidance is provided on removing the dependency on JenOS.

1 Introduction

This document provide basic guidelines for migrating previously developed ZigBee PRO applications to the stack provided in the NXP JN517x ZigBee 3.0 SDK (JN-SW-4270). These migration guidelines are intended for users who wish to migrate existing applications to ZigBee 3.0 for the JN517x platforms, where these applications were developed from either of the following:

- ZigBee 3.0 for the JN516x platforms (based on the JN-SW-4170 SDK)
- ZigBee Light Link or Home Automation for the JN516x platforms (based on the JN-SW-4168, JN-SW-4067 or JN-SW-4062 SDK)

The ZigBee Light Link and Home Automation SDKs incorporate the legacy NXP ZigBee PRO network stack that uses the NXP proprietary operating system, JenOS, which is designed around statically assigned resources that are configured using Eclipse-based configuration diagrams.

The NXP ZigBee 3.0 stack is designed to run without any dependencies on an operating system. The components within the JN517x ZigBee 3.0 SDK provide a breadth of tools to allow easy migration from JenOS-based legacy ZigBee Home Automation/Light Link applications to the ZigBee 3.0 standard.

Note that the non-OS resources that were included in JenOS (e.g. Persistent Data Manager) are included in the JN51xx Core Utilities (JCU), supplied in the JN517x ZigBee 3.0 SDK.

2 Legacy JenOS-based Architecture

The ZigBee PRO Stack (ZPS) interfaces to the IEEE802.15.4 MAC layer to achieve the 802.15.4 wireless communication. The stack exports Application Programming Interface (API) functions to enable the user of the stack to send ZigBee PRO data over the 802.15.4 wireless medium.

The stack is notified of various events from the MAC by the MAC interrupt which is handled in the **zps_isrMAC** Interrupt Service Routine (ISR) provided by the JenOS interrupt handler. These events are serialised based on their MAC message type using two JenOS queues, namely zps_msgMlmeDcfmInd and zps_msgMcpsDcfmInd.

The processing of events on these queues is done in the context of the **zps_taskZPS** task. In addition to these queues, the stack also consumes Timer Server (TSV) events which are serialised through the zps_msgTimeEvents queue. The TSV events are generated in response to internal deadlines that are set and used within the ZigBee PRO stack.

 © NXP Semiconductors 2016

The ZigBee PRO stack communicates to the user application by posting messages on a shared queue, APP_msgZpsEvents, using an application task which can be configured through the Eclipse-based configuration. At least one task must be provided by the application to handle events generated by the stack.

This sets up two producer/consumer hierarchies:

- ZPS task as the consumer and the MAC ISR/TSV timer as producer
- Application as the consumer and the ZPS task as the producer

The ZPS task has a lower base priority than the application task. However, it locks a shared mutex which raises the priority of the ZPS task to be the dispatch priority of the mutex group. It pushes any pending event to the application queue. It also briefly lowers its priority to its base priority, which enables the application task attached to the queue to run and read the message off the queue. This allows the shared queue to be small. Once the message is consumed by the application task and the function returns, the ZPS task captures the shared mutex which allows it to run in the dispatch priority, allowing it to run to completion.

There is no queue between the application and the stack to serialise transmission API function calls originating in the application. This is achieved by using mutexes (mutexZPS and mutexMAC) to share the use of the MAC and protect the context structures within the stack.

The features of JenOS include:

- Task functions are allowed to run to completion.
- Tasks can be activated by explicit API function calls.
- Tasks can be scheduled to be in a pending state as a result of OS software timer expiry, posting a message on the queue or task activation in the currently running task.
- Stack memory is shared to store stack frames for all OS tasks. This results in optimisation of stack usage by running tasks in co-operative groups.
- Scheduling priorities of tasks are controlled by being part of the mutex group.
- All the OS resources are statically allocated at compile-time.

A hierarchy exists in the processing of messages by the ZPS task from the three queues. The timer queue and the MCPS data queue are given higher precedence than the MLME Queue.

The ZPS task reads the queued message from the timer/MCPS queue every time the task is activated. It grabs the ZPS mutex which makes it the highest priority task in the mutex group of which the application task is also a member. This allows the ZPS task to run un-interrupted when it is processing the queued message. The only events that can interrupt the ZPS task are higher priority tasks which are not part of the mutex group and interrupts.

The advantages and disadvantages of this arrangement are as follows:

Advantages

- Lower memory requirements for the application queue (needs to be only of size 1)
- Low over-head for context switches, as tasks can be run co-operatively
- Lower stack usage as a result of few context switches and shared stack
- Flexible approach because higher priority tasks can be outside the co-operative group, so reduces blocking time

 © NXP Semiconductors 2016

Disadvantages

- Increases the need for bigger queues for timer/MCPS/MLME queues
- Critical messages can be lost as a result of queue overflows

3 Application Migration to Non-OS-based ZigBee 3.0

This section describes how to migrate a JN516x application designed using the OS-based ZigBee Home Automation/Light Link SDK to a non-OS-based ZigBee 3.0 application for a JN517x device. It is recommended that to migrate to ZigBee 3.0, the approach should be to develop a new application to which existing application functionality will be ported.

The effort can be broadly divided into the following:

1. Pre-requisite changes
2. Replacing the ISRs
3. Replacing Application Task with equivalent functions
4. Replacing the OS timers with equivalents
5. Creation of queues
6. Implementing critical sections

These areas are detailed in the sub-sections below.

3.1 Application Pre-requisites

The application controls the various resources the ZigBee 3.0 stack uses.

All internal stack communication with the radio and timers is synchronised through three queues:

- MLME Queue (Service MLME data, i.e. for association, beacons etc)
- MCPS Queue (Service MCPS data requests)
- Timer Queue (Service timer expiry)

The size of these queues depends on the size of the network and can be customised depending on the application.

The following resources are mandatory and should always be assigned in the application:

```
/*#defines*/
#define TIMER_QUEUE_SIZE            8
#define MLME_QUEUE_SIZE             8
#define MCPS_QUEUE_SIZE             24


/*imported variables*/
extern tszQueue zps_msgMlmeDcfmInd;
extern tszQueue zps_msgMcpsDcfmInd;
extern tszQueue zps_TimeEvents;



zps_tsTimeEvent asTimeEvent[TIMER_QUEUE_SIZE];
MAC_tsMcpsVsDcfmInd asMacMcpsDcfmInd[MCPS_QUEUE_SIZE];
MAC_tsMlmeVsDcfmInd  asMacMlmeVsDcfmInd[MLME_QUEUE_SIZE];
```

 © NXP Semiconductors 2016

The application must implement the function to allocate the various resources that it uses and those required by the stack, as follows:

```
void app_vInitResources(void)
{

    /* Initialise the Z timer module */
  ZTIMER_eInit(asTimers, sizeof(asTimers) / sizeof(ZTIMER_tsTimer));

  /* Create Z timers */

  /* create all the queues*/

        ZQ_vQueueCreate(&zps_msgMlmeDcfmInd,MLME_QUEUE_SIZE,sizeof(MAC_tsMlm
        eVsDcfmInd),(uint8*)asMacMlmeVsDcfmInd);

        ZQ_vQueueCreate(&zps_msgMcpsDcfmInd,MCPS_QUEUE_SIZE,sizeof(MAC_tsMcp
        sVsDcfmInd),(uint8*)asMacMcpsDcfmInd);

        ZQ_vQueueCreate(&zps_TimeEvents,TIMER_QUEUE_SIZE,sizeof(zps_tsTimeEv
        ent),(uint8*)asTimeEvent);

        /* initialise any other resource the application uses */

}
```

The ZigBee 3.0 stack does not have any dependency on an OS. It is designed to function in a single non-terminating loop. After coming out of a cold restart, the application should implement and call the hardware initialisation function. The hardware initialisation function makes use of the standard NXP application hardware peripheral interface.

The example below provided for 3 general interrupt sources, the base band controller (MAC interrupt), the UART0 and the system controller interrupt.

```
/* define interrupt priorities here – Higher value equates to higher priority*/
#define NVIC_INT_PRIO_LEVEL_SYSCTRL (3)
#define NVIC_INT_PRIO_LEVEL_BBC     (9)
#define NVIC_INT_PRIO_LEVEL_UART0   (4)


  void app_vSetUpHardware(void)
  {
       vAHI_Uart0RegisterCallback ( APP_isrUart );
       vAHI_SysCtrlRegisterCallback ( APP_isrSysController );
       u32AHI_Init();
       vAHI_InterruptSetPriority(MICRO_ISR_MASK_BBC,
NVIC_INT_PRIO_LEVEL_BBC );
       vAHI_InterruptSetPriority(MICRO_ISR_MASK_UART0,
NVIC_INT_PRIO_LEVEL_UART0 );
      vAHI_InterruptSetPriority(MICRO_ISR_MASK_SYSCTRL,
  NVIC_INT_PRIO_LEVEL_SYSCTRL );
  }
```

 © NXP Semiconductors 2016

After hardware initialisation, the control of the application should be passed to the control scheduling loop which may be implemented in its simplest form as follows:

```
void app_vMainloop(void)
{
        /*     idle task commences on exit from OS start call */
        while (TRUE) {
                /* application functions */
                zps_taskZPS();
                ZTIMER_vTask();

                /* kick the watchdog timer */
                vAHI_WatchdogRestart();
                PWRM_vManagePower();

        }
}
```

The stack communicates with the application though a callback. The callback should be implemented as below:

```
PUBLIC void APP_vGenCallback(uint8 u8Endpoint, ZPS_tsAfEvent *psStackEvent)
{


}
```

An application may implement queues to break context between transmitting and receiving stack events. How the application chooses to service the events is beyond the scope of these migration guidelines.

If the application implements mutex locks, the mutex shall be implemented as a function call, as illustrated below:

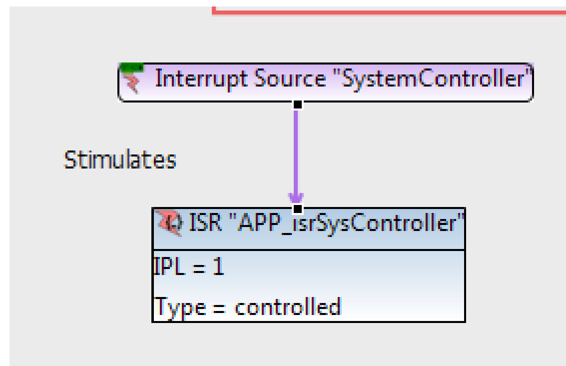
```
bool_t bAppMutex = FALSE;

PUBLIC void* APP_vMutex(void)
{
        return &bAppMutex;
```

 © NXP Semiconductors 2016

3.2 Replace ISRs and Exception Handler

In an OS-based application, each interrupt is configured in the OS diagram, as shown in the following example.



The removal of the OS allows use of the JN517x Integrated Peripheral APIs to handle exceptions and interrupt. The interrupt service routines are registered as callbacks to the relevant integrated peripheral handlers.

For example:

```
vAHI_Uart0RegisterCallback( APP_isrUart );
```

This registers a callback `APP_isrUart` with the integrated peripheral interrupt handler for UART0. The callback function must then be implemented in the application. Please refer to the *JN517x Integrated Peripherals API User Guide (JN-UG-3118)* for further details.

The minimum Interrupt Priority Level (IPL) is 1 and the maximum IPL is 15.

To set the priority of an interrupt:

```
vAHI_InterruptSetPriority(MICRO_ISR_MASK_BBC,     NVIC_INT_PRIO_LEVEL_BBC );
```

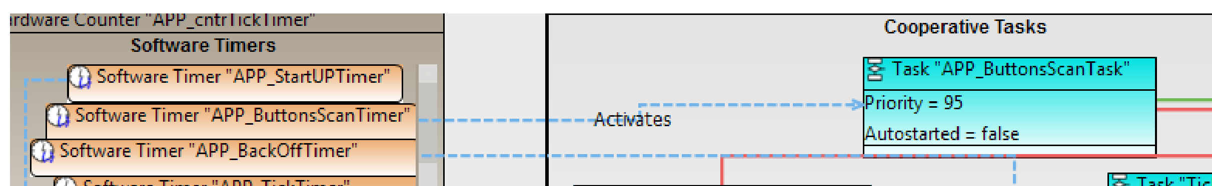
This function is documented in the above User Guide (JN-UG-3118).

The exception handlers are provided in the SDK and the source is available in **ZigbeeCommon/Source/port_JN51x.c**.

3.3 Replace OS tasks

In OS-based application design, a task is invoked by the expiry of a timer or a message posted in a message queue. In the non-OS-based design, the tasks must be re-implemented as functions. These functions may then be called as a result of polling the message queue (for new messages) or as the result of the expiry of a timer.

This non-OS-based design has a mechanism for installing the task as a callback function for timer expiry. For example, in the diagram below, the APP_ButtonScanTimer expiry activates APP_ButtonScanTask.



© NXP Semiconductors 2016

Replace the OS_TASK as a function. For example, replace

```
OS_TASK(APP_ButtonScanTask)
{
}
```

by

```
PUBLIC void APP_ButtonScanTask(void* pv)
{
}
```

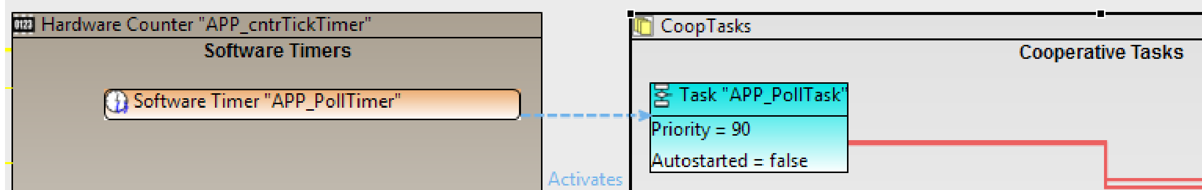
This can be achieved either by editing the code for each function or by making a selective redefinition of the macro OS_TASK in a common header file. An example of the latter is:

```
#ifdef NO_RTOS
#define OS_TASK(a) PUBLIC void ##a(void *pv)
#endif
```

 © NXP Semiconductors 2016

3.4 Replace OS Timers and Callback Handler

In an OS-based application, the software timers are initialised in the OS configuration. An example of APP_PollTimer is shown below, where APP_PollTimer activates APP_PollTask.



The timer is started using the **OS_eStartSWTimer()** function. Upon timer expiry, the corresponding task is activated. The timer can also be stopped by using the **OS_eStopSWTimer()** function.

In a non-OS-based application, the ZTimer utility can be used to initialise, start and stop the timer. The ZTimer utility is part of the ZigBee 3.0 SDK. The ZTimer module can be initialised using the **ZTIMER_eInit()** function and a timer callback is registered using the **ZTIMER_eOpen()** function.

e.g. `ZTIMER_eOpen(&u8TimerPoll, APP_cbTimerPoll, NULL, ZTIMER_FLAG_PREVENT_SLEEP);`

Then this timer can be started using the **ZTIMER_eStart()** function.

e.g. `ZTIMER_eStart(u8TimerPoll, u32Time);`

Once the timer expires, the registered callback function (e.g. APP_cbTimerPoll) will be called.

If needed, the application can also stop the timer using the **ZTIMER_eStop()** function.

The **ZTIMER_vTask()** function should be called in the scheduling loop. Note that the ZTimer uses the tick timer interrupt to provide the timer functionality. The expiry happens in the context of the interrupt but the expiry callback function is only called in the context of the **ZTIMER_vTask()** function.

3.5 Create Message Queues

In an OS-based application, the message queues are created by the OS configuration.

For a non-OS-based application, the message queues are created by calling the **ZQ_vQueueCreate()** function for all the required message queues in application, as follows.

```
ZQ_vQueueCreate (
                &APP_msgEvents,
                 APP_QUEUE_SIZE,
                 sizeof(APP_tsEvent),
                 (uint8*)asAppMsgEvent);
```

where:

- `APP_msgEvents` is a structure of type **tszQueue**
- `asAppMsgEvent` is a buffer for the queue of size APP_QUEUE_SIZE

 © NXP Semiconductors 2016

To send a message, the OB-based function call needs to be replaced with the non-OS equivalent call, as shown in the example below:

Replace

```
OS_ePostMessage(APP_msgSerialTx, &u8TxByte)
```

with

```
ZQ_bQueueSend(&APP_msgSerialTx, &u8TxByte)
```

Similarly, to receive a message on the queue, replace all occurrences of the OS-based function call with its non-OS equivalent, as shown in the example below:

Replace

```
OS_eCollectMessage(APP_msgSerialTx, &u8Byte)
```

with

```
ZQ_bQueueReceive(&APP_msgSerialTx, &u8Byte)
```

In the OS-based application design, there are tasks that are invoked once a message queue has received a message. In a non-OS-based application, the queue can be checked for incoming message repeatedly - if a message is available then the desired function is called.

For example:

```
if (ZQ_bQueueReceive (&APP_msgPOREvents, &sAppPorEvent))
{
        DBG_vPrintf(TRACE_LIGHT_NODE, "\n\nPOR Event: %d", sAppPorEvent.eType);
        vHandlePorEvent(&sAppPorEvent);
}
```

The code can be called repeatedly to check for any message received on the APP_msgPOREvents queue.

The Queue functions are interrupt-safe and do not require additional protection from interrupts.

3.6 Implementing Critical Sections

In the OS-based application design, a mutex (mutual exclusion) provides control of shared resources and also prevents interrupt protection. The mutex group takes the highest priority of the tasks allowed to lock the mutex, and hence provides an implementation of priority ceiling and priority inheritance.

A mutex is locked and unlocked using the critical section API functions. A critical section of code within the user task/ISR must be delimited with the **OS_eEnterCriticalSection()** and **OS_eExitCriticalSection()** functions.

In the non-OS equivalent, the functions **ZPS_u8GrabMutexLock()** and **ZPS_u8ReleaseMutexLock()** are used to delimit the critical section of code.

 © NXP Semiconductors 2016

Note the following:

- Critical sections disable interrupts. They should be used for very critical data handling where being pre-emptied by an interrupt would cause data corruption. In the critical section, another task with priority below 11 would not pre-empt the processing. The limit of 11 is arbitrary, so you can still have higher priority interrupts. As this switches off interrupts, this should only be used for fast, urgent and critical code.
- A mutex can be used for any code which should not be re-entered. So you can use this for any shared modules or frequently used API functions which you do not want to be pre-empted or called out of context. You must provide your own mutex handle which needs to be created in your application as a function call.

It should also be noted that critical sections only provide protection from interrupts with priority lower than 11. The OSMIUM nested interrupt functionality will ensure that higher priority interrupts always pre-empt lower priority interrupts. When a critical section is entered, the code which is running in the critical section will be given an interrupt priority of 11. Hence, it is vital that critical sections are used where this latency is taken into account and acceptable.

4 Migration of NXP Application Notes

This section describes how to migrate the following NXP Application Notes (provided for ZigBee Light Link and Home Automation) to ZigBee 3.0:

- JN-AN-1171: ZigBee Light Link Solution
- JN-AN-1189: ZigBee Home Automation Demonstration

4.1 Migrating JN-AN-1171 (ZigBee Light Link Solution)

For ZigBee 3.0, it is recommended that you use the following Application Notes instead of migrating JN-AN-1171:

- JN-AN-1218: ZigBee 3.0 Light Bulbs
- JN-AN-1219: ZigBee 3.0 Controller and Switch

If it is really necessary to migrate JN-AN-1171, please follow the guidelines below. The ZigBee 3.0 Application Notes JN-AN-1218 and JN-AN-1219 also serve as good references while doing the migration.

1. Touchlink functionality is now part of the Base Device in ZigBee 3.0. Therefore, the application does not need the Touchlink code (**app_light_commissioning_task.c** file). Touchlink and the other required Base Device features can be enabled by including the following build options in the Makefile.

    a. `BDB_SUPPORT_TOUCHLINK_INITIATOR_END_DEVICE ?= 1`

       Add this option on Controller builds to enable Touchlink Initiator functionality on ZigBee End Devices.

    b. `BDB_SUPPORT_TOUCHLINK_INITIATOR_ROUTER ?= 1`

       This is only required if a ZigBee Router needs Touchlink Initiator capability.

    c. `BDB_SUPPORT_TOUCHLINK_TARGET ?= 1`

       Add this option on Light builds to enable Touchlink Target functionality on ZigBee Router devices.

 © NXP Semiconductors 2016

d. `BDB_SUPPORT_NWK_STEERING ?= 1`

   This must be enabled to be compliant with ZigBee 3.0 Base Device.

e. `BDB_SUPPORT_NWK_FORMATION ?= 1`

   If this is enabled then a ZigBee Router will be capable of forming a distributed network. In the case of a ZigBee Co-ordinator, this will enable the functionality for forming a centralised network.

f. `BDB_SUPPORT_FIND_AND_BIND_INITIATOR ?= 1`

   Add if 'Finding and Binding' initiator functionality is needed in the application.

g. `BDB_SUPPORT_FIND_AND_BIND_TARGET ?= 1`

   Add if 'Finding and Binding' target functionality is needed in the application.

h. `BDB_SUPPORT_OOBC ?= 1`

   Add if 'Out of Band Commissioning' functionality is needed in the application.

Add "`include $(SDK_BASE_DIR)/Components/BDB/Build/config.mk`" in the Makefile so that the appropriate Base Device source will be taken while building the application.

`STACK_SIZE` and `MINIMUM_HEAP_SIZE` must be defined in the Makefile. Also remove **APP_stack_size.ld** from the application's build directory.

Set `JENNIC_SDK` to `JN-SW-4270` and `JENNIC_STACK` to `ZCL` in the Makefile.

Replace `APP_CLUSTER_ZLL_SRC` with `APP_CLUSTER_LIGHTING_SRC`.

Update `UTIL_SRC_DIR` and `BDB_SRC_DIR` with the following.

```
UTIL_SRC_DIR = $(COMPONENTS_BASE_DIR)/ZigbeeCommon/Source
BDB_SRC_DIR = $(COMPONENTS_BASE_DIR)/BDB/
```

Remove all APPSRC files starting with **os_**.

In the Makefile, rename "`Utilities/Include`" to "`ZigbeeCommon/Include`", due to the folder structure alignment in ZigBee 3.0.

The old ZCL structure is split into ZCL and ZCIF, hence include "`ZCIF/Include`" in the Makefile include section.

For any further change needed, please refer to the Makefiles present in the Application Notes JN-AN-1219 (ZigBee 3.0 Controller and Switch) and JN-AN-1217 (ZigBee 3.0 Base Device).

2. Remove all the OS-related includes and function calls from the application. During application initialisation **(APP_vInitialiseNode()**), the Base Device must be initialised by calling the **BDB_vInit()** API function. For further details of the BDB API functions and their usage, please refer to the *ZigBee 3.0 Devices User Guide (JN-UG-3114)*.
3. All the events from the underlying layers (Stack and Base Device) are passed to a callback function defined in the application. Hence, the application needs to define the callback function **APP_vBdbCallback()** and handle the events within this callback. For the events coming into this callback, refer to the *ZigBee 3.0 Devices User Guide (JN-UG-3114)*. For reference implementations, refer to the JN-AN-1219 and JN-AN-1217 Application Notes.

© NXP Semiconductors 2016

4. In a ZigBee 3.0 application design, after the initialisation process has completed, the **BDB_vStart()** function is called before entering into **APP_vMainLoop**. The function **BDB_vStart()** will start the Base Device.
5. The application must include the **bdb_options.h** file to configure various aspects of the Base Device by altering Base Device attributes as well as constants in manufacturer-specific applications.
6. To start the Touchlink initiator process, the **BDB_vZclEventHandler()** function needs to be called with event type `BDB_E_ZCL_EVENT_TL_START`.
7. From the ZCL callback, on receiving the GENERAL_CLUSTER_ID_IDENTIFY event, call **BDB_vZclEventHandler()** with the event type BDB_E_ZCL_EVENT_IDENTIFY_QUERY. This is required by the 'Finding and Binding' handling inside the Base Device.
8. From the ZCL callback, on receiving the event COMMISSIONING_CLUSTER_ID_TL_COMMISSIONING, if the event type is interPAN and related to the ZLL Profile, call **BDB_vZclEventHandler()** with the event type set to BDB_E_ZCL_EVENT_TL_IPAN_MSG. This is needed by the Base Device to proceed with the Touchlink state machine.
9. If the network leave event for self-leave originated when Touchlink process is ongoing, call **BDB_vZclEventHandler()** with event type ZPS_EVENT_NWK_LEAVE_CONFIRM.

4.2 Migrating JN-AN-1189 (ZigBee Home Automation Demo)

For ZigBee 3.0, it is recommended that you use the following Application Notes instead of migrating JN-AN-1189:

- JN-AN-1218: ZigBee 3.0 Light Bulbs
- JN-AN-1219: ZigBee 3.0 Controller and Switch
- JN-AN-1220: ZigBee 3.0 Sensors

If it is really necessary to migrate JN-AN-1189, please follow the guidelines below. The ZigBee 3.0 Application Notes JN-AN-1218, JN-AN-1219 and JN-UG-1220 also serve as good references while doing the migration.

1. Network formation, network joining and 'Find and Binding' are now part of the Base Device in ZigBee 3.0. Hence, the application no longer needs code related to this functionality and the code should be removed. This can be done by removing the files **haEzJoin.c/.h**, **haEzFindAndBind.c/.h** and **haKeys.c/.h**. Network formation, network joining, 'Finding and Binding' and other required Base Device features can be enabled by including the following build options in the Makefile.

   a. `BDB_SUPPORT_NWK_STEERING ?= 1`

      This must be enabled to be compliant with the ZigBee 3.0 Base Device.

   b. `BDB_SUPPORT_NWK_FORMATION ?= 1`

      If this is enabled then a ZigBee Router will be capable of forming a distributed network. In the case of a ZigBee Co-ordinator, this will enable the functionality for forming a centralised network.

   c. `BDB_SUPPORT_FIND_AND_BIND_INITIATOR ?= 1`

      Add if 'Finding and Binding' initiator functionality is needed in the application.

d. `BDB_SUPPORT_FIND_AND_BIND_TARGET ?= 1`

Add if 'Finding and Binding' target functionality is needed in the application.

e. `BDB_SUPPORT_OOBC ?= 1`

Add if 'Out of Band Commissioning' functionality is needed in the application.

Add "`include $(SDK_BASE_DIR)/Components/BDB/Build/config.mk`" in the Makefile so that the appropriate Base Device source will be taken while building the application.

`STACK_SIZE` and `MINIMUM_HEAP_SIZE` must be defined in the Makefile. Also remove **APP_stack_size.ld** from application's build directory.

Set `JENNIC_SDK` to `JN-SW-4270` and `JENNIC_STACK` to `ZCL` in the Makefile.

Replace `APP_CLUSTER_MEASUREMENT_AND_SENSING` with `APP_CLUSTER_MEASUREMENT_AND_SENSING_SRC`.

Update `UTIL_SRC_DIR` and `BDB_SRC_DIR` with the following:

```
UTIL_SRC_DIR = $(COMPONENTS_BASE_DIR)/ZigbeeCommon/Source
BDB_SRC_DIR = $(COMPONENTS_BASE_DIR)/BDB/
```

Remove all APPSRC files starting with **os_**.

In the Makefile, rename "`Utilities/Include`" to "`ZigbeeCommon/Include`", due to the folder structure alignment in ZigBee 3.0.

The old ZCL structure is split into ZCL and ZCIF, hence include "`ZCIF/Include`" in the Makefile include section.

For any further change needed, please refer to the Makefiles present in the ZigBee 3.0 Application Notes JN-AN-1218, JN-AN-1219 and JN-AN-1220.

2. As the child ageing mechanism is now part of the ZigBee R21 stack, any code related to child ageing must be removed from the JN-AN-1189 Application Note. This includes the removal of the files **AgeChildren.c/.h** and **PingParent.c/.h**.
3. Remove all the OS-related includes and function calls from the application. During application initialisation **(APP_vInitialiseNode()**), the Base Device must be initialised by calling the **BDB_vInit()** API function. For further details of the BDB API functions and their usage, please refer to the *ZigBee 3.0 Devices User Guide (JN-UG-3114)*. Any references to OS-related resources, including **os.h** and **os_gen.h**, must also be removed.
4. All the events from the underlying layers (Stack and Base Device) are passed to a callback function defined in the application. Hence, the application needs to define the callback function **APP_vBdbCallback()** and handle the events within this callback. For the events coming into this callback, refer to the *ZigBee 3.0 Devices User Guide (JN-UG-3114)*. For reference implementations, refer to the JN-AN-1218, JN-AN-1219 and JN-AN-1220 Application Notes.
5. In a ZigBee 3.0 application design, after the initialisation process has completed, the **BDB_vStart()** function is called before entering into **APP_vMainLoop**. The function **BDB_vStart()** will start the Base Device.
6. The application must include the **bdb_options.h** file to configure various aspects of the Base Device by altering Base Device attributes as well as constants in manufacturer-specific applications.

© NXP Semiconductors 2016

Please refer to the *ZigBee 3.0 Devices User Guide (JN-UG-3114)* for information on how to trigger network formation, joining and 'Finding and Binding' on ZigBee 3.0 devices.

4.3 Working with Multiple Eclipse Environment

Application development for the JN517x devices is performed in LPCXpresso, which is an Eclipse-based IDE but is different from BeyondStudio used for JN516x application development. To make sure the same code base can be supported in either development environment, it is recommended that separate projects files are created for each chip family and hence IDE.

The following procedures in Section 4.3.1 and Section 0 will help you to create these project files and enable the use of the same source code for different chip variants. You will need to follow both procedures, in BeyondStudio for JN516x and in LPCXpresso for JN517x.

4.3.1 JN516x Project for BeyondStudio

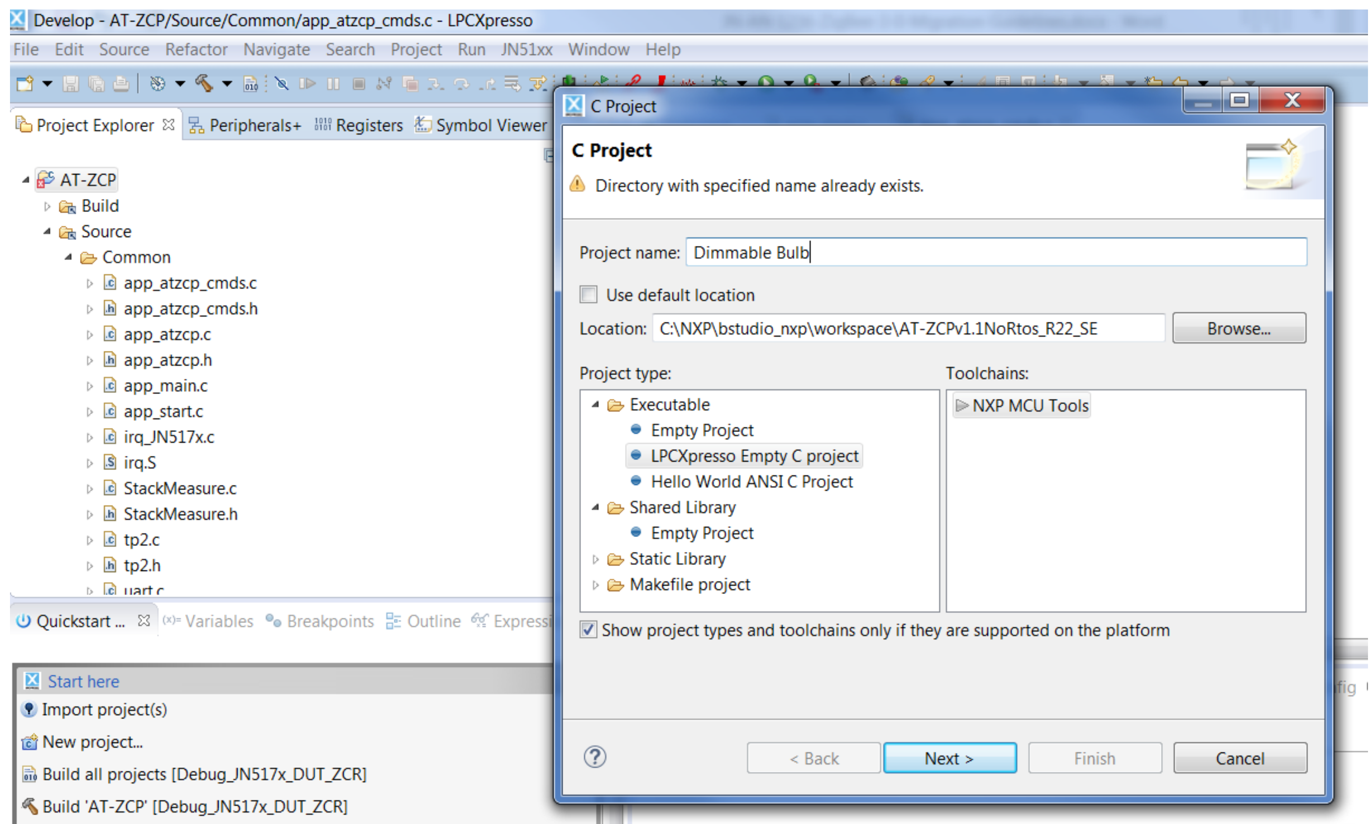
1. Start Windows Explorer and follow the path to the location of the project folder. Under the project folder, create a new project folder, **JN516x**.
2. Open BeyondStudio and remove any existing projects that you have imported.
3. You already have the JN516x project files, so copy your **.project** and **.cproject** files into your **JN516x** folder.

Note: The build path is now one folder lower. For example, if the build path was `${ProjDirPath}/Build/DeviceUnderTest` then it should now be `${ProjDirPath}/../Build/DeviceUnderTest`.

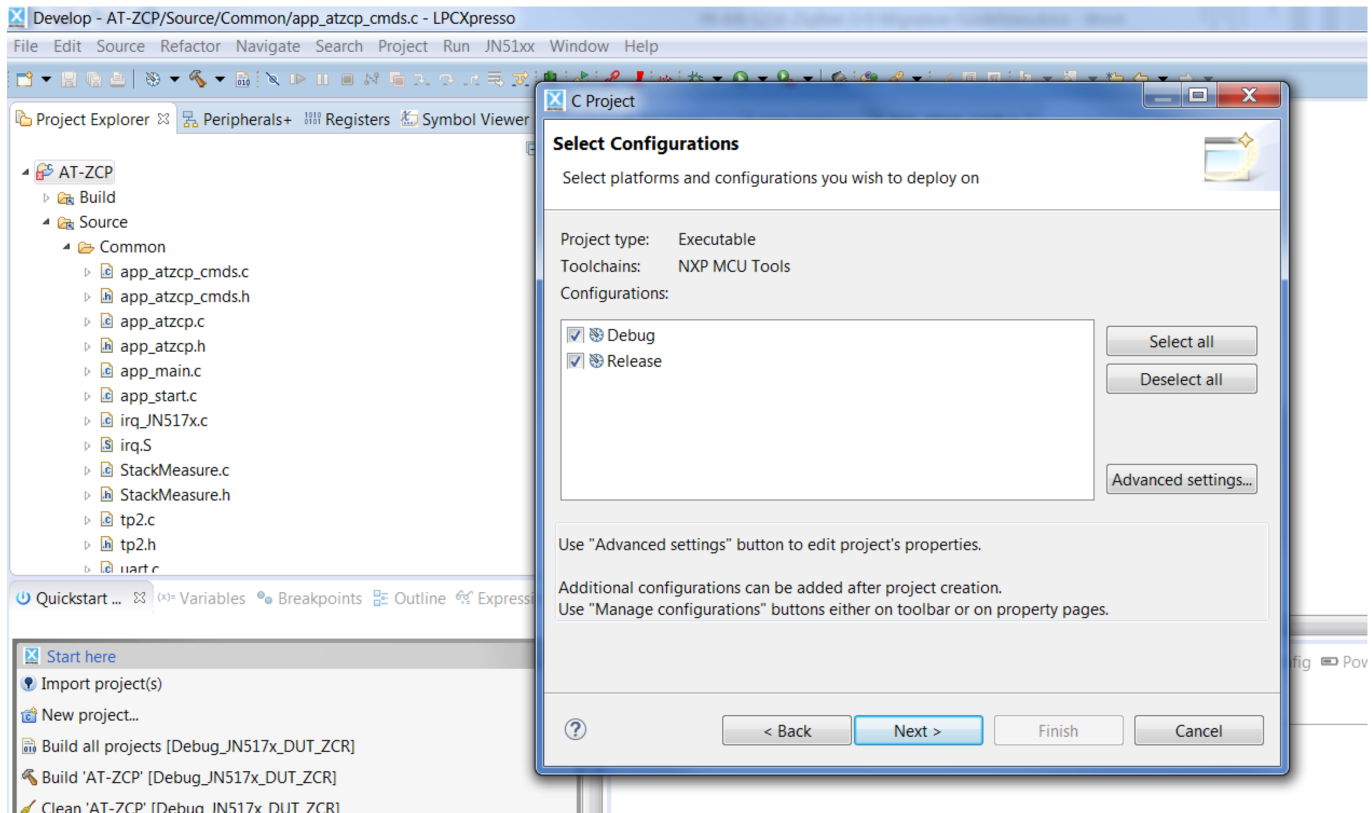
 © NXP Semiconductors 2016

4.3.2 JN517x Project for LPCXpresso

1. Start Windows Explorer and follow the path to the location of the project folder. Under the project folder, create a new project folder, **JN517x**.
2. Open LPCXpresso and remove any existing projects that you have imported.
3. Create a C project in LPCXpresso as follows:
   a) Right-click on the left-hand **Project Explorer** pane.
   b) Select the menu option **New > C Project** – the **C Project** screen appears.
   c) Fill in the relevant project name (e.g. Dimmable Bulb) in the **C Project** screen.
   d) Select the location and set the toolchain to NXP MCU Tools, then click **Next**.

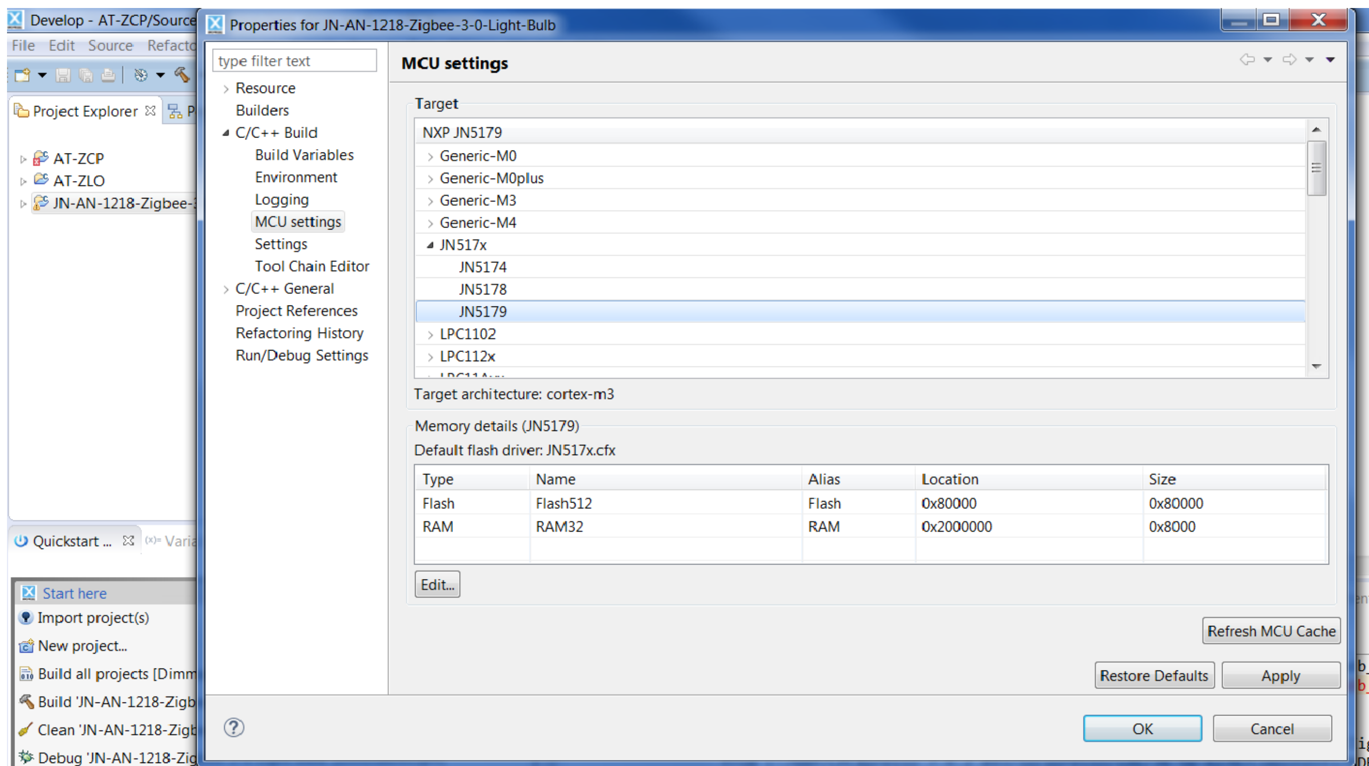


4. On the **Select Configurations** screen, select the Debug and Release configurations, then click **Next** – this should create these two build configurations.



© NXP Semiconductors 2016

5. Select the MCU chip family for the debugger as follows:
   a) In the **Project Explorer** pane, right-click on the project and select **Properties** from the drop-down menu.
   b) In the resulting **Properties** screen, navigate to **MCU settings** under **C/C++ Build.**
   c) In the **Target** pane, select the target chip type (e.g. JN5179).
   d) Click **Apply** and then **OK.**

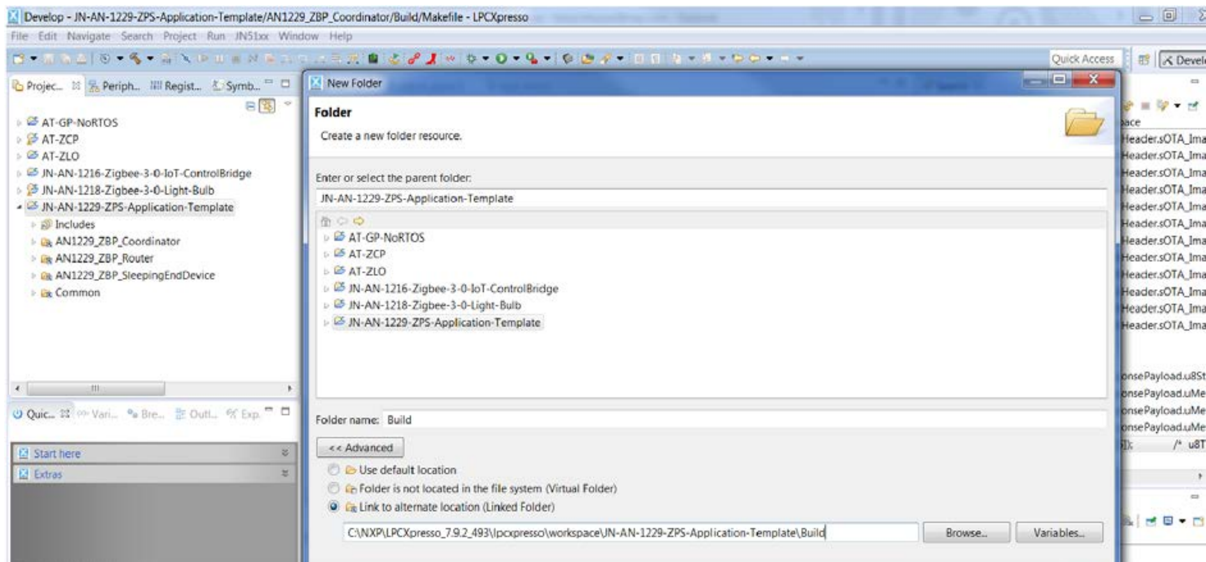


6. Once the project is created, create the link to your source, as follows.

   a) In the **Project Explorer** pane, select and right-click on the newly created project (e.g. Dimmable Bulb).

   b) From the drop-down menu, select **New > Folder** and on the resulting screen click **Advanced**.

   c) Select the radio button **Link to alternate location** and then select the source folders or build folders that you normally have in your project. This creates a link to the existing file system location rather than create a new location, so that editing it in one build configuration will automatically edit it in the other one.



7. Create a multiple build configuration. To do this, right-click on the **Project Explorer** pane, then select **Properties >C/C++ Build** and edit the configuration. In LPCXpresso, you need to select the relevant toolchain, which should be NXP MCU Tools - this is done by selecting **Tool Chain Editor** under **C/C++ Build**. You can add a new build configuration using the **Manage Configuration** button and then edit the selected configuration, rename it or create a new configuration based on an existing one.

Note: The build path is now one folder lower. For example, if the build path was `${ProjDirPath}/Build/DeviceUnderTest` then it should now be `${ProjDirPath}/../Build/DeviceUnderTest`.

4.4 Changes to the Makefiles

The JN517x chips have a different CPU core from the previous JN51xx chips. The Makefile should reflect the change in chip family and device-specific build and compile options.

The required CPU stack size and heap size need to be defined using the following variables:

```
STACK_SIZE             = 6000
MINIMUM_HEAP_SIZE      = 2000
```

Any reference to **APP_STACK_SIZE.ld** must be removed.

The Makefile must define:

```
ifeq ($(JENNIC_CHIP_FAMILY),JN517x)
JENNIC_SDK ?= JN-SW-4270
ENDIAN ?= LITTLE_ENDIAN_PROCESSOR
endif
```

Link time optimisation must be disabled and hence the following flag must be defined:

```
DISABLE_LTO=1
```

The exception handlers can be pulled into the build using:

```
APPSRC += port_JN517x.c
```

The registration of the custom MAC interrupt handler must also be done, as shown below:

```
ifeq ($(JENNIC_CHIP_FAMILY),JN517x)
INTERRUPT_HANDLER_BBC = zps_isrMAC
LDFLAGS += -Wl,-u$(INTERRUPT_HANDLER_BBC) -Wl,-
defsym,vAHI_IntHandlerBbc_select=$(INTERRUPT_HANDLER_BBC)
endif
```

The legacy application builds required each function to have its own section and required these sections to be pulled explicitly into the build. The following lines in the Makefile:

```
$(TARGET)_$(JENNIC_CHIP)$(BIN_SUFFIX).bin:
$(TARGET)_$(JENNIC_CHIP)$(BIN_SUFFIX).elf
	$(info Generating binary ...)
	$(OBJCOPY) -j .version -j .bir -j .flashheader -j .vsr_table -j
.vsr_handlers -j .rodata -j .text -j .data -j .bss -j .heap -j .stack -S -O binary
$< $@
```

must be changed to:

```
$(TARGET)_$(JENNIC_CHIP)$(BIN_SUFFIX).bin:
$(TARGET)_$(JENNIC_CHIP)$(BIN_SUFFIX).elf
	$(info Generating binary ...)
	$(OBJCOPY) -S -O binary $< $@
```

 © NXP Semiconductors 2016

5 Related Documents

The following manuals will be useful in developing custom applications based on this Application Note:

- ZigBee 3.0 Stack User Guide [JN-UG-3113]
- ZigBee 3.0 Devices User Guide [JN-UG-3114]
- ZigBee Cluster Library (for ZigBee 3.0) User Guide [JN-UG-3115]
- JN51xx Core Utilities User Guide [JN-UG-3116]
- JN517x Integrated Peripherals API User Guide [JN-UG-3118]
- JN517x LPCXpresso Installation and User Guide [JN-UG-3109]

All the above manuals are available as PDF documents from the ZigBee 3.0 page of the NXP web site.

 © NXP Semiconductors 2016

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 Semiconductors

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

www.nxp.com

© NXP Semiconductors 2016