Application Note: JN-AN-1085

JenNet Tutorial

This Application Note provides a tutorial on developing a JenNet wireless network application using the Jenie API (Application Programming Interface). The tutorial can be followed using a JN5148 or JN5139 evaluation kit.

The following steps are covered in the tutorial:

1. **Development tools and basic application structure**
2. **Starting the network with a Co-ordinator node**
3. **Joining Router nodes to the network and sending data**
4. **Registering, discovering, binding and sending data using services**
5. **Joining sleeping End Device nodes to the network**

Introduction

This tutorial describes JenNet software development for both JN5148 and JN5139 devices. Software development differs between these devices in that JN5148 development makes use of the Eclipse environment while JN5139 development uses the Code::Blocks environment. Where necessary, certain sections of this document provide information for JN5148 Eclipse development only and others for JN5139 Code::Blocks development only - you should ensure that you follow the sections appropriate for the device type with which you are working.

Despite the differences in the development environment used to compile the software, the same source code is compiled to form both the JN5148 and JN5139 applications.

To follow this tutorial, a working knowledge of C programming is assumed. A visual differencing tool, such as WinMerge, may be useful to compare the differences between the code at various stages of the tutorial.

Whilst this tutorial provides introductory information on JenNet application development, the following documentation provides more details on the topics covered in this tutorial:

- **JN-UG-3041: JenNet Stack User Guide** provides concept and practical information on JenNet, and provides function reference information for the Jenie API
- **JN-RM-2003: Board API Reference Manual** details the API for interacting with the evaluation kit boards, including the functions for controlling the LEDs and buttons
- **JN-UG-3007: JN51xx Flash Programmer User Guide** provides instructions on using the Flash Programmer tool

JN5148 Development Hardware and Software

To follow this tutorial for JN5148 devices, you will need the following kit:

- **JN5148-EK010 Evaluation Kit**

The JN5148 Software Developer's Kit (SDK), which includes the Eclipse IDE, is used to develop software for JN5148 devices. This SDK can be installed by downloading and running the following installers from the Jennic website:

- **JN-SW-4041: JN5148 SDK Toolchain**
- **JN-SW-4040: JN5148 SDK Libraries**

Whilst this tutorial provides basic information on the use of Eclipse, additional information on the installation of the SDK and use of Eclipse can be found in the following documents, available on the Jennic website:

- **JN-UG-3064: JN5148 SDK Installation Guide** provides instructions for installing the JN5148 SDK
- **JN-UG-3063: Eclipse IDE User Guide** provides information on using the Eclipse IDE

In order to build the software provided with this Application Note, its folder must be placed directly under **C:\Jennic\Application\**. This directory is automatically created when you install the SDK Toolchain.

JN5139 Development Hardware and Software

To follow this tutorial for JN5139 devices, you will need one of the following kits:

- **JN5139-EK000 IEEE 802.15.4/JenNet Evaluation Kit** (no longer available)
- **JN5139-EK010 ZigBee Evaluation Kit** (no longer available)

The JN5139 Software Developer's Kit (SDK), which includes the Code::Blocks IDE, is used to develop software for JN5139 devices. This SDK can be installed by downloading and running the following installers from the Jennic website:

- **JN-SW-4031: SDK Toolchain**
- **JN-SW-4030: SDK Libraries**

In order to build the software provided with this Application Note, its folder must be placed directly under **C:\Jennic\cygwin\jennic\SDK\Application\**. This directory is automatically created when you install the SDK Toolchain.

Whilst this tutorial provide basic information on the use of Code::Blocks, additional information on the installation of the SDK and use of Code::Blocks can be found in the following documents, available on the Jennic website:

- **JN-UG-3035: SDK Installation Guide** provides instructions for installing the JN5139 SDK
- **JN-UG-3028: Code::Blocks IDE User Guide** provides information on using the Code::Blocks IDE

Step 1 – Development Tools and Application Structure

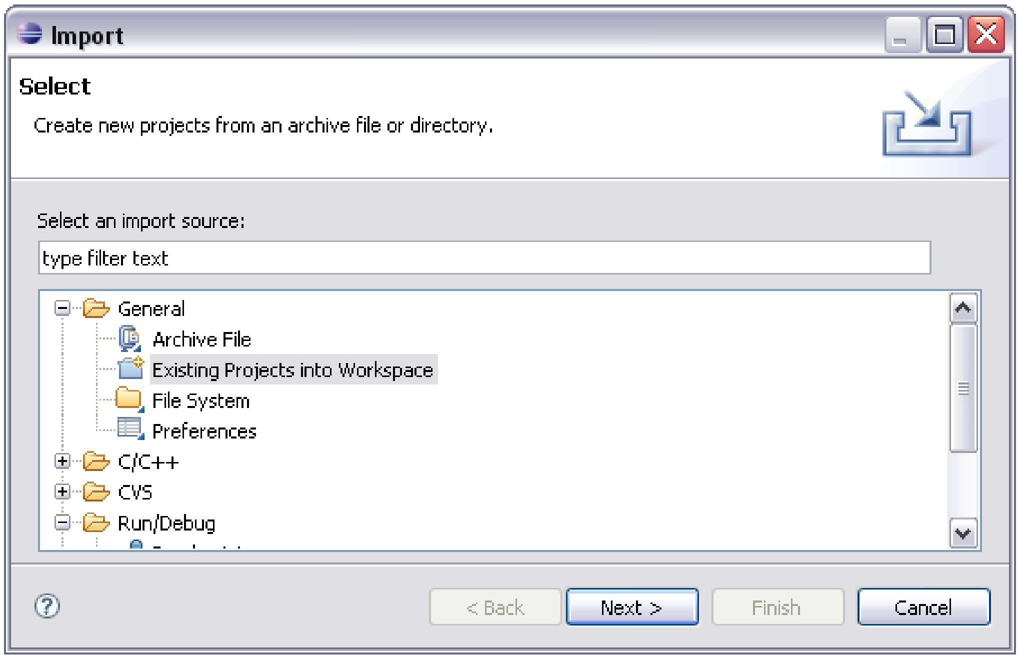
This step introduces the Eclipse and Code::Blocks IDEs, as well as associated project files and compilation. The basic structure of a JenNet application is also outlined. There are separate sections to introduce the JN5148 Eclipse IDE and JN5139 Code::Blocks IDE - follow the appropriate sections for the device type with which you are working.

Step 1.1 – JN5148 Eclipse IDE Introduction

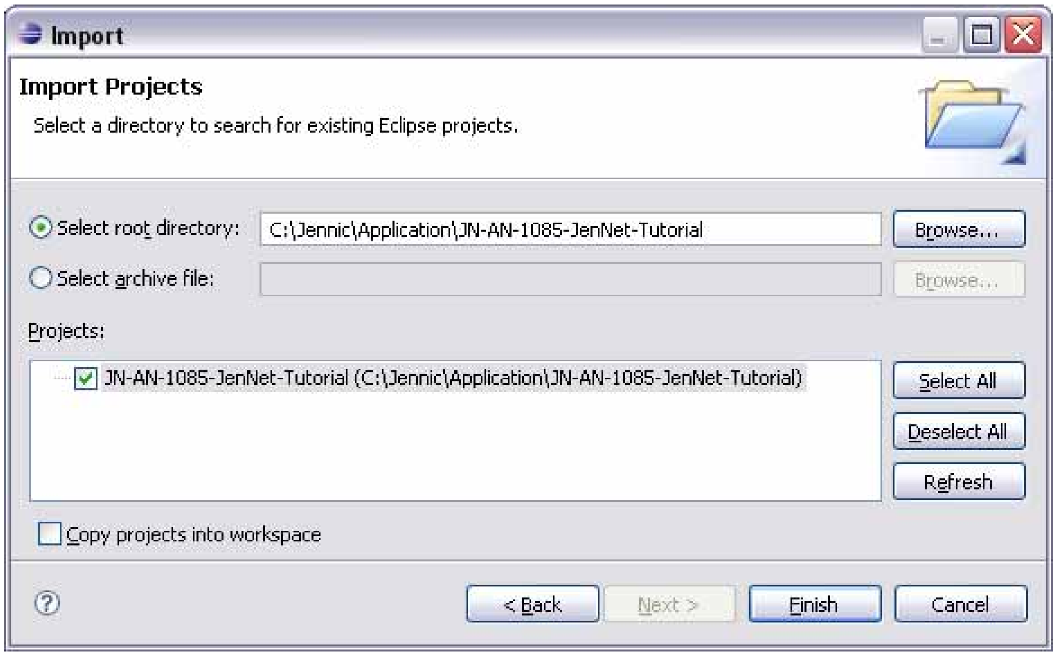
The JN5148 SDK Toolchain builds applications using **make** and makefiles. **Make** can be instigated from the command line or using the Eclipse IDE. For this tutorial, we will use the Eclipse IDE to build our applications.

The project and configuration settings used by Eclipse are specified using the **.project** and **.cproject** files located in the application folder. These settings and the source files should first be imported into Eclipse. This is initiated by selecting the **File > Import…** menu entry in the main Eclipse window.

In the **Import** window that opens, select the **General > Existing Projects into Workspace** entry in the tree. Then click the **Next** button to progress to the next stage.

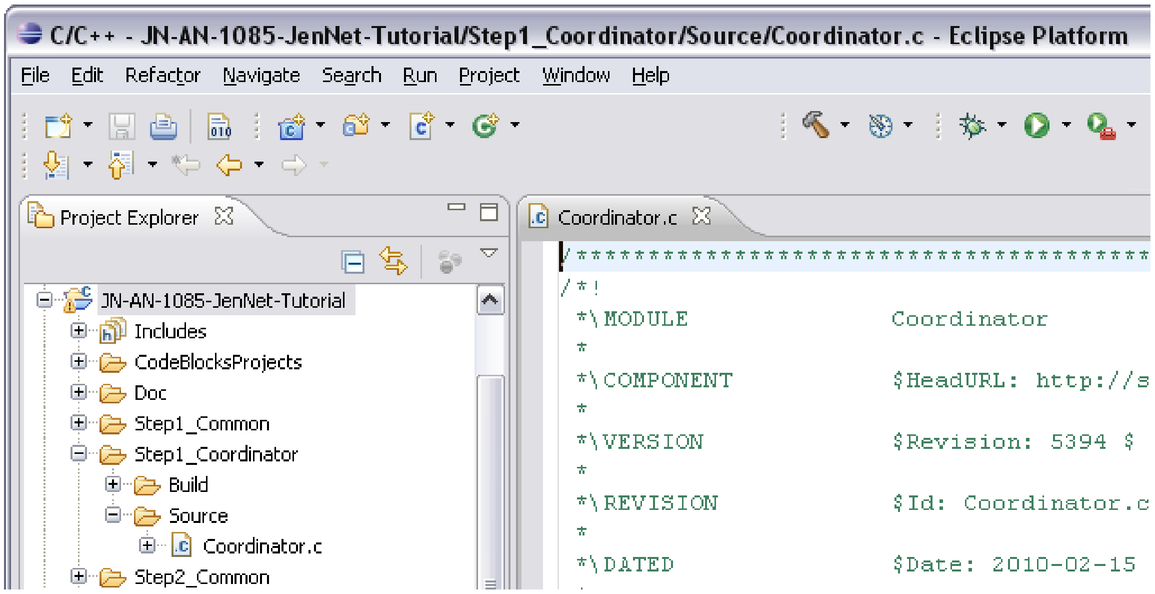


The **Import** window will then update. Press the **Browse** button adjacent to the **Select root directory** radio button, and navigate to and select the **C:\Jennic\Application\JN-AN-1085-JenNet-Tutorial** folder. Finally, press **Finish** to complete the task of importing the project into Eclipse.


Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory: C:\Jennic\Application\JN-AN-1085-JenNet-Tutorial
Browse...
Select archive file:
Browse...
Projects:
JN-AN-1085-JenNet-Tutorial (C:\Jennic\Application\JN-AN-1085-JenNet-Tutorial)
Select All
Deselect All
Refresh
Copy projects into workspace
< Back
Next >
Finish
Cancel

To the left of the window, the **Project Explorer** pane lists all the projects in the Eclipse workspace. The project for this tutorial will be named **JN-AN-1085-JenNet-Tutorial**.

The source files for each project are grouped under it in a tree structure that mirrors the file system of the application folder that was imported. The tree branches can be opened by clicking the "+" boxes at each level, allowing the source files to be explored. The image below shows the opened file **Step1_Coordinator\Source\Coordinator.c**.



```
/****************************************
/*!
 *\MODULE              Coordinator
 *
 *\COMPONENT           $HeadURL: http://s
 *
 *\VERSION             $Revision: 5394 $
 *
 *\REVISION            $Id: Coordinator.c
 *
 *\DATED               $Date: 2010-02-15
```

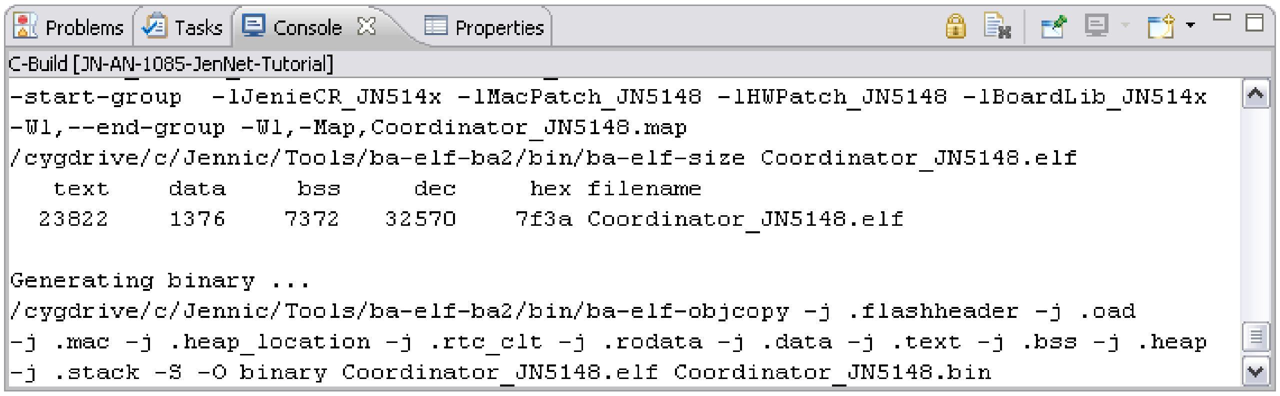
The source files for the Step 1 Co-ordinator node are stored in the following folders in the application folder:

- **Step1_Common\Source** contains files that are shared by applications for all device types. In Step 1, there is only a single device type, the Co-ordinator, but later steps that add additional device types will use the **StepX_Common** folder for shared files.
- **Step1_Coordinator\Source** contains files that are specific to the Co-ordinator application for Step 1.

Eclipse uses a 'build target' (also known in Eclipse as a Configuration) to build an individual application, with each build target specifying which makefile is used to build each application. The build targets used in this tutorial will have been imported as part of the project configuration files. The makefile **Step1_Coordinator\Build\makefile** is used to build the Coordinator device type for this step of the tutorial. In order to add new source files for compilation, the appropriate makefile should be edited to include any additional files.

Pre-compiled binary files are included with the Application Note to allow experimentation without requiring the compilation of source files. The binary for the Step 1 Co-ordinator application is **Step1_Coordinator\Build\Coordinator_JN5148.bin**.

Just to make sure everything is correct, build the Step 1 Coordinator application by ensuring the **JN-AN-1085-JenNet-Tutorial** project is selected in **Project Explorer**, click the 'build' button and select the **Step1_Coordinator** build target from the drop down. The **Console** tab in the lower part of the window should displaythe final "Generating binary" message with no errors (see the image below) and the newly compiled binary should overwrite the pre-compiled binary included in the Application Note.



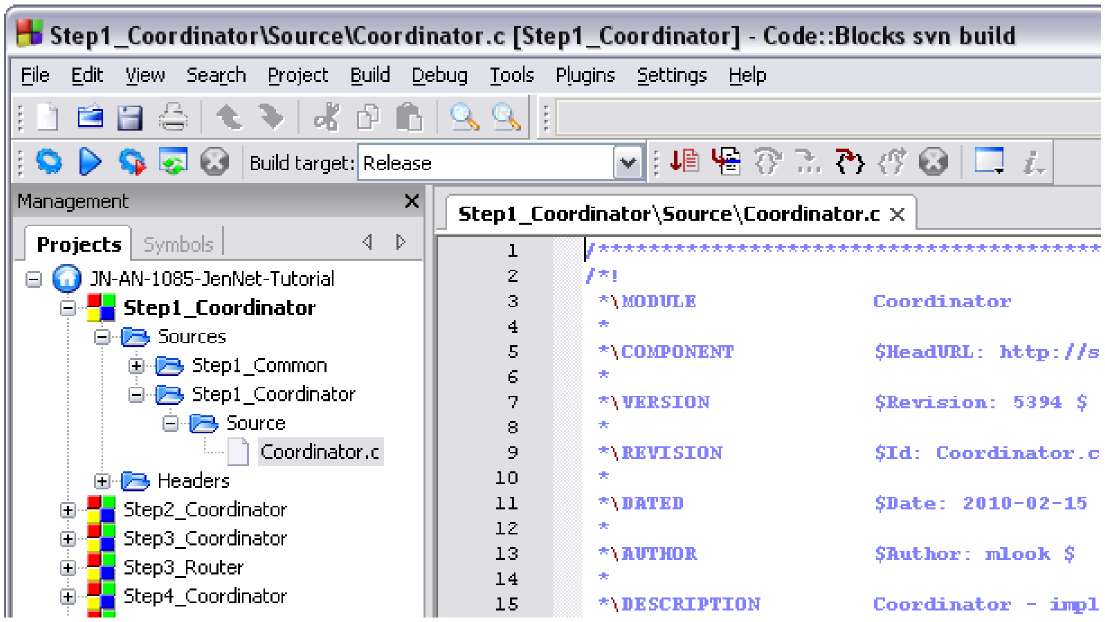
C-Build [JN-AN-1085-JenNet-Tutorial]

```
-start-group  -lJenieCR_JN514x -lMacPatch_JN5148 -lHWPatch_JN5148 -lBoardLib_JN514x
-Wl,--end-group -Wl,-Map,Coordinator_JN5148.map
/cygdrive/c/Jennic/Tools/ba-elf-ba2/bin/ba-elf-size Coordinator_JN5148.elf
   text    data     bss     dec     hex filename
  23822    1376    7372   32570    7f3a Coordinator_JN5148.elf

Generating binary ...
/cygdrive/c/Jennic/Tools/ba-elf-ba2/bin/ba-elf-objcopy -j .flashheader -j .oad
-j .mac -j .heap_location -j .rtc_clt -j .rodata -j .data -j .text -j .bss -j .heap
-j .stack -S -O binary Coordinator_JN5148.elf Coordinator_JN5148.bin
```

In this step, we will not cover downloading and running the compiled application on the evaluation kit boards, as this application does not perform any useful tasks and these activities are covered in Step 2.

Step 1.1 – JN5139 Code::Blocks IDE Introduction


Step1_Coordinator\Source\Coordinator.c [Step1_Coordinator] - Code::Blocks svn build
File Edit View Search Project Build Debug Tools Plugins Settings Help
Build target: Release
Management
Projects Symbols
JN-AN-1085-JenNet-Tutorial
Step1_Coordinator
Sources
Step1_Common
Step1_Coordinator
Source
Coordinator.c
Headers
Step2_Coordinator
Step3_Coordinator
Step3_Router
Step4_Coordinator
Step1_Coordinator\Source\Coordinator.c

```
/******************************************
/*!
 *\MODULE              Coordinator
 *
 *\COMPONENT           $HeadURL: http://s
 *
 *\VERSION             $Revision: 5394 $
 *
 *\REVISION            $Id: Coordinator.c
 *
 *\DATED               $Date: 2010-02-15
 *
 *\AUTHOR              $Author: mlook $
 *
 *\DESCRIPTION         Coordinator - impl
```

The JN5139 SDK Toolchain provides options to build applications either using **make** from the command line or using the Code::Blocks IDE. For this tutorial, we will use the Code::Blocks IDE to build our application (though makefiles are also provided for those wishing to explore further).

The source code and libraries that are used to build an application are specified using a Code::Blocks project file, which has extension **.cbp**. All the project files for this tutorial can be found in the **CodeBlocksProjects** folder within the application folder. The project files are named to indicate the tutorial step and device type that they compile.

Code::Blocks project files may be grouped together into a workspace, allowing related projects to be opened together in the Code::Blocks IDE. All the project files for this tutorial have been grouped together in the **JN-AN-1085-JenNet-Tutorial.workspace** file, also located in the **CodeBlocksProjects** folder.

Double-clicking the file **CodeBlocksProjects\JN-AN-1085-JenNet-Tutorial.workspace** will open the Code::Blocks IDE and load all the project files used in this tutorial. If double-clicking does not work, open Code::Blocks from the Windows **Start** menu, and use **File>Open** in Code::Blocks to select and open the workspace file. The Code::Blocks IDE should appear as shown above (once the **Coordinator.c** file has been opened):

Note: Project and workspace files can be associated with Code::Blocks by using the **Settings>Environment** menu entry.

It is also possible to open the individual **.cbp** project files.

To the left of the window, the open Code::Blocks project files are listed. The project for Step 1 Co-ordinator is named **Step1_Coordinator**. Code::Blocks operates on a currently active project, indicated by the name displayed in bold type. If **Step1_Coordinator** is not the currently active project, it can be activated by right-clicking it and selecting **Activate project** from the displayed menu - its name should then be displayed in bold.

The source files for each project are grouped under it in a tree structure. The tree branches can be opened by clicking the "+" boxes at each level, allowing the source files to be explored. Double-clicking a source file leaf in the tree will open the file for editing in the IDE. The image above shows the opened file **Step1_Common\Source\Coordinator.c**.

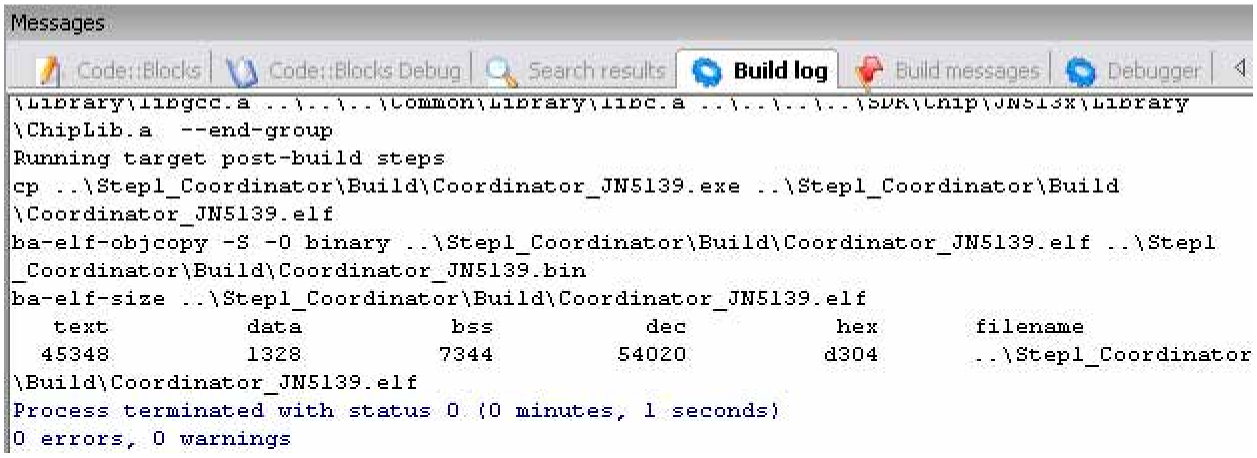
The source files for the Step 1 Co-ordinator node are stored in the following folders within the application folder:

- **Step1_Common\Source** contains files that will are shared by applications for all device types. In Step 1, there is only a single device type, the Co-ordinator, but later steps that add additional device types will use the **StepX_Common** folder for shared files.
- **Step1_Coordinator\Source** contains files that are specific to the Co-ordinator application for Step 1.

New source files are added to a project using the **Project>Add Files…** menu entry and using the standard file selection dialogue. The **Project** menu contains many common IDE tasks related to the active project (shown in bold) in the **Projects** list. The same menu can be accessed by right-clicking a project in the **Projects** list.

Pre-compiled binary files are included with the Application Note to allow experimentation without requiring the compilation of source files. The binary for the Step 1 Co-ordinator application is **Step1_Coordinator\Build\Coordinator_JN5139.bin**.

Just to make sure everything is correct, try clicking the 'build' button to build the application. The 'rebuild' button can also be used to rebuild all source files in the application rather than only the updated files. The **Build log** tab in the lower part of the window should contain no errors (see the image below) and the newly compiled binary should overwrite the pre-compiled binary included in the Application Note.



```
\Library\libgcc.a ..\..\..\Common\Library\libc.a ..\..\..\..\SDK\Chip\JN513x\Library
\ChipLib.a  --end-group
Running target post-build steps
cp ..\Step1_Coordinator\Build\Coordinator_JN5139.exe ..\Step1_Coordinator\Build
\Coordinator_JN5139.elf
ba-elf-objcopy -S -O binary ..\Step1_Coordinator\Build\Coordinator_JN5139.elf ..\Step1
_Coordinator\Build\Coordinator_JN5139.bin
ba-elf-size ..\Step1_Coordinator\Build\Coordinator_JN5139.elf
   text          data          bss          dec          hex       filename
  45348          1328          7344        54020         d304       ..\Step1_Coordinator
\Build\Coordinator_JN5139.elf
Process terminated with status 0 (0 minutes, 1 seconds)
0 errors, 0 warnings
```

In this step, we will not cover downloading and running the compiled application on the evaluation kit boards, as this application does not perform any useful tasks, and these activities are covered in Step 2.

Step 1.2 – Basic Application Structure

The Jenie API consists of a set of functions that can be called by the application to make use of JN5148 and JN5139 facilities and perform wireless networking operations.

The Jenie API also calls a number of callback functions that must be present in the application. These are used to initialise the networking environment, to allow the application's main tasks to be run, and to inform the application of networking and hardware events that are occurring. These callback functions therefore form the core of a JenNet application.

Coordinator.c

The **Coordinator.c** file, located in the **Step1_Coordinator\Source** folder, provides the starting point of the network Co-ordinator node application.

Coordinator.c includes a number of header files, as follows:

```
#include <jendefs.h>   /* Standard Jennic type definitions */
#include <Jenie.h>     /* Jenie API definitions and interface */
#include "App.h"       /* Application definitions and interface */
```

Coordinator.c declares a global array of data for storing a Routing table. The Routing table is used internally by JenNet to route data messages through the network. The application is responsible for allocating this memory, allowing the memory used to be tailored to the expected size of the network. The number of elements in the Routing table must match the maximum expected number of nodes in the network.

The Routing table should not be used in any way by the application other than allocating the memory and providing the JenNet stack with a pointer to the memory.

```
/* Routing table storage */
PRIVATE tsJenieRoutingTable asRoutingTable[ROUTING_TABLE_SIZE];
```

This file also contains the Jenie API callback functions, listed and described below:

vJenie_CbConfigureNetwork

```
PUBLIC void vJenie_CbConfigureNetwork(void)
{
    /* Set up routing table */
    gJenie_RoutingEnabled       = TRUE;
    gJenie_RoutingTableSize     = ROUTING_TABLE_SIZE;
    gJenie_RoutingTableSpace    = (void *) asRoutingTable;
    /* Change default network config */
    gJenie_NetworkApplicationID = APPLICATION_ID;
    gJenie_PanID                = PAN_ID;
    gJenie_Channel              = CHANNEL;
    gJenie_ScanChannels         = SCAN_CHANNELS;
}
```

This callback function is called only when the application is performing a cold start, on power-up. The application should set the global variables that provide details of the Routing table memory, identify the network and specify the radio channels on which the network may operate:

- **`gJenie_RoutingEnabled`** specifies that the provided Routing table should be used.
- **`gJenie_RoutingTableSize`** specifies how many elements are in the Routing table array.

- `gJenie_RoutingTableSpace` provides a pointer to the memory allocated by the application for JenNet to store the Routing table array.
- `gJenie_NetworkApplicationID` is used to identify the JenNet application by specifying the Network Application ID. This identifier allows different JenNet network applications to co-exist on the same radio channel in the same physical operating space.
- `gJenie_PanID` is used to specify the PAN (Personal Application Network) ID to be used by the IEEE 802.15.4 networking layer. However, the Co-ordinator will "listen" to neighbouring wireless networks and if it finds that the specified PAN ID is already in use by another network, it will increment its own PAN ID until it finds a value that is unique.
- `gJenie_Channel` specifies the 2400-MHz radio channel to be used for communications, as a number in the range 11 to 26. This variable can alternatively be set to zero to allow the Co-ordinator to choose the quietest channel automatically.
- `gJenie_ScanChannels` specifies a bit mask of channels to scan when searching for the quietest channel when `gJenieChannel` is set to zero.

For the purposes of this tutorial, these global variables are set to fixed values defined in **App.h**.

vJenie_CbInit

```
PUBLIC void vJenie_CbInit(bool_t bWarmStart)
{
    teJenieStatusCode eStatus; /* Jenie status code */

    /* Initialise application */
    vApp_CbInit(bWarmStart);

    /* Start Jenie */
    eStatus = eJenie_Start(E_JENIE_COORDINATOR);
}
```

This callback function is called on both a cold start and a warm start, and is generally used to initialise the application.

The **vJenie_CbInit()** function ends by calling **eJenie_Start()**. This function instructs JenNet to start running as a Co-ordinator, which will then try to create the network for other nodes to join.

Note: For the purpose of this tutorial, all node types in the network will use the same application with only slight differences. Therefore, we have put the core code of our applications in the file **App.c**, located in the **Step1_Common\Source** folder, which is referenced from the application on each node type. This file contains a set of callback functions that mirror those found in the Jenie API and that the Jenie API functions will call. For example, as shown in the code fragment above, the function **vJenie_CbInit()** calls the equivalent function **vApp_CbInit()** in the file **App.c**.

vJenie_CbMain

```
PUBLIC void vJenie_CbMain(void)
{
	/* Call application main function */
	vApp_CbMain();
}
```

This function is called regularly by the JenNet stack to allow the application to perform its main processing. This code should be non-blocking. The function simply calls the equivalent function in **App.c** (see Note above).

vJenie_CbStackMgmtEvent

```
PUBLIC void vJenie_CbStackMgmtEvent(teEventType eEventType, void *pvEventPrim)
{
	/* Pass event on to application */
	vApp_CbStackMgmtEvent(eEventType, pvEventPrim);
}
```

This callback function is called whenever a network event occurs, such as the network being joined or lost. The type of event is provided along with further information dependent upon the event type. Again, this function simply calls the equivalent function in **App.c**.

vJenie_CbStackDataEvent

```
PUBLIC void vJenie_CbStackDataEvent(teEventType eEventType, void *pvEventPrim)
{
	/* Pass event on to application */
	vApp_CbStackDataEvent(eEventType, pvEventPrim);
}
```

This function is called whenever a data event occurs, such as data or a data ack being received. The type of event is provided along with further information, including any data received, dependent upon the event type. Once again, this function simply calls the equivalent function in **App.c**.

vJenie_CbHwEvent

```
PUBLIC void vJenie_CbHwEvent(uint32 u32DeviceId, uint32 u32ItemBitmap)
{
	/* Pass event on to application */
	vApp_CbHwEvent(u32DeviceId, u32ItemBitmap);
}
```

This function is called whenever an event occurs on one of the JN5139 hardware peripherals, such as the on-chip timers. The device causing the event is indicated, along with further information on the cause of the event from the device. Yet again, this function simply calls the equivalent function in **App.c**.

Events are queued within the JenNet stack, as they occur in an interrupt context. The main thread then removes the events from the queue by calling this function. Therefore, this function is called outside the interrupt context. It is possible to specify interrupt callbacks for many device events, which will then be called from within the interrupt context, if this is required for an application.

App.c and App.h

As explained in the Note in the above section, the applications we create for the different node types will all perform the same task. To simplify development, the core code for our application will be created in the files **App.c** and **App.h**, located in the **Step1_Common\Source** folder.

- **App.h** contains the definitions for the various network IDs and the declarations of the public functions in **App.c**.
- **App.c**, at this stage, simply contains empty callback functions mirroring those of the Jenie API.

Step 2 – Starting the Network on the Co-ordinator

Each network needs to have a single Co-ordinator node. The Co-ordinator is responsible for starting the network for all other devices to join. Since the Co-ordinator performs such a crucial role, it is permanently powered.

This step uses the stack management events to determine when the network is started on the Co-ordinator node. A flashing LED is used to provide feedback, which in turn is controlled by a timer. Additionally, the serial port on UART0 is used to output JenNet stack events and function calls to aid understanding of how the JenNet stack is used.

The process of downloading the application from the PC to the JN5148 or JN5139 device is also covered.

The files required for this step are located in the **Step2_Common\Source** and **Step2_Coordinator\Source** folders.

Step 2.1 – Code Changes

The following changes have been made to the Step 1 code:

Printf.c and Printf.h

These files (installed as part of the SDK rather than this Application Note) are used to provide access to the Printf functions that output to UART0. These functions are used to provide a display of the JenNet stack events and function calls made in the application.

The files have been added to the Code::Blocks project file and Eclipse makefile to ensure that they are compiled into the application.

Coordinator.c

Header Files

The following header files have been added to access functions in other modules:

```
#include <JPI.h>                    /* Jenie Peripheral Interface {v2} */
#include <AppHardwareApi.h>         /* Hardware peripherals not in JPI.h {v2} */
#include <Printf.h>                 /* Basic Printf to UART0-19200-8-NP-1 {v2} */
#include <LedControl.h>             /* Led Interface {v2} */
```

vJenie_CbConfigureNetwork

Code is included to initialise UART0 for use with the Printf functions, then **vPrintf()** is used to output the initial message indicating that the **vJenie_CbConfigureNetwork()** function has been called.

```
/* Open UART for printf use {v2} */
vUART_printInit();
/* Output function call to UART */
vPrintf("\nvJenie_CbConfigureNetwork()\n");
```

vJenie_CbInit

When a device sleeps, any UARTs in use are disabled and need to be re-initialised when a warm start takes place. While this step of the tutorial covers a Co-ordinator node, which does not sleep, this code is essential for the End Device node introduced in Step 5. Code is included in the **vJenie_CbInit()** function to detect a warm start and re-initialise UART0 for use with the Printf functions. **vPrintf()** is then used to indicate that the call to **vJenie_CbInit()** has occurred.

```
/* Warm start - reopen UART for printf use {v2} */
if (bWarmStart) vUART_printInit();
/* Output function call to UART */
vPrintf("vJenie_CbInit(%d)\n", bWarmStart);
```

The function ends by outputting the call to **eJenie_Start()** and its returned status code to UART0.

```
/* Output function call to UART {v2} */
vPrintf("eJenie_Start(COORDINATOR) = %d\n", eStatus);
```

The tutorial will continue to make liberal use of **vPrintf()** to output similar diagnostic messages for other stack events and function calls.

App.c

This section describes how the **App.c** file for Step 1 has been modified to produce the **App.c** file for Step 2.

Header Files

The following header files have been added:

```
#include <JPI.h>              /* Jenie Peripheral Interface {v2} */
#include <Printf.h>           /* Basic Printf to UART0-19200-8-NP-1 {v2} */
#include <LedControl.h>       /* LED Interface {v2} */
```

Functions declared in **Printf.h** will continue to be used for diagnostic feedback and functions from **LedControl.h** will be used to control the LEDs on the evaluation kit boards.

Global Variables

The following global variables have been added:

```
PRIVATE bool_t   bNetworkUp;  /* Network running {v2} */
PRIVATE uint8  au8Led[2];     /* LED states {v2} */
PRIVATE uint8   u8Tick;       /* Ticker {v2} */
```

- **`bNetworkUp`** will be used to check whether the network is currently up and running.
- **`au8Led`** will be used to set a state for each of the two LEDs:
  - A value of 0 will be used to indicate OFF
  - A value of 0xFF will be used to indicate ON
  - Other values will be used to flash the LED at various speeds
- **`u8Tick`** is a simple incrementing timer that will be updated by the tick timer 100 times per second - it will be used to flash the LEDs. The tick timer is automatically configured by JenNet to tick 100 times per second, providing a convenient timer for use by an application.

vApp_CbInit

```
PUBLIC void vApp_CbInit(bool_t bWarmStart)
{
    /* Is this a cold start ? */
    if (! bWarmStart)
    {
        /* Network won't be up */
        bNetworkUp = FALSE;
    }

    /* Initialise LEDs */
    vLedInitRfd();

    /* Turn off LEDs */
    au8Led[0] = 0;
    au8Led[1] = 0;
}
```

This function is used to initialise the application:

1. **bNetworkUp** is initialised to FALSE if a cold start is occurring.
2. The LED hardware is initialised and the states set to OFF.

vApp_CbMain

```
PUBLIC void vApp_CbMain(void)
{
        /* Regular watchdog reset */
        #ifdef WATCHDOG_ENABLED
                vAHI_WatchdogRestart();
        #endif

        /* Network is down ? */
        if (! bNetworkUp)
        {
                /* Flash LED0 quickly while we wait for the network to start */
                au8Led[0] = 0x02;
        }
        /* Led has been left flashing ? */
        else if (au8Led[0] != 0 && au8Led[0] != 0xFF)
        {
                /* Turn off LED */
                au8Led[0] = 0x00;
        }
}
```

This function is called regularly by the stack.

JN5148 device features on-chip watchdog hardware that will reset the chip if not regularly serviced. As this function is frequently called by the stack, the watchdog is serviced here.

The current state of **bNetworkUp** is monitored:

- If the network is not running, LED 0 is flashed quickly.
- If the network is running and LED 0 is flashing, LED 0 is extinguished (set to OFF).

vApp_CbStackMgmtEvent

```
PUBLIC void vApp_CbStackMgmtEvent(teEventType eEventType, void *pvEventPrim)
{
    /* Which event occurred ? */
    switch (eEventType)
    {
       /* Indicates stack is up and running */
       case E_JENIE_NETWORK_UP:
       {
              /* Output to UART */
              vPrintf("vApp_CbStackMgmtEvent(NETWORK_UP)\n");
              /* Network is now up */
              bNetworkUp = TRUE;
       }   break;

       /* Indicates stack has reset */
       case E_JENIE_STACK_RESET:
       {
              /* Output to UART */
              vPrintf("vApp_CbStackMgmtEvent(STACK_RESET)\n");
              /* Network is now down */
              bNetworkUp = FALSE;
       }   break;

       default:
       {
              /* Unknown data event type */
       }   break;
    }
}
```

This function handles the stack management events.

- `E_JENIE_NETWORK_UP` occurs in the Co-ordinator when the network is started.
- `E_JENIE_STACK_RESET` occurs if the network is lost.

`bNetworkUp` is updated according to the event that occurs, in order to track the status of the network. **vPrintf()** is used to indicate the events that are occurring.

vApp_CbHwEvent

```
PUBLIC void vApp_CbHwEvent(uint32 u32DeviceId,uint32 u32ItemBitmap)
{
    uint8   u8Led;

    /* Is this the tick timer ? */
    if (u32DeviceId == E_JPI_DEVICE_TICK_TIMER)
    {
        /* Increment our ticker */
        u8Tick++;

        /* Loop through LEDs */
        for (u8Led = 0; u8Led < 2; u8Led++)
        {
            /* Set LED according to status */
            if (au8Led[u8Led] == 0 || au8Led[u8Led] == 0xFF)
                vLedControl(u8Led, au8Led[u8Led]);
            else
                vLedControl(u8Led, au8Led[u8Led] & u8Tick);
        }
    }
}
```

This function is called whenever a hardware event occurs.

When called as a result of the tick timer ticking, **`u8Tick`** is incremented. The LEDs are then looped through and are set or cleared according to their status:

- If the LED is OFF or ON, it is simply controlled (set to OFF or ON, according to the LED status).
- If the LED is flashing, a bitwise AND is performed between the LED status and **`u8Tick`**, and the LED is controlled according to the result. By specifying different bit masks, the LEDs can be made to flash at different speeds.

Step 2.2 – Building and Downloading the Application

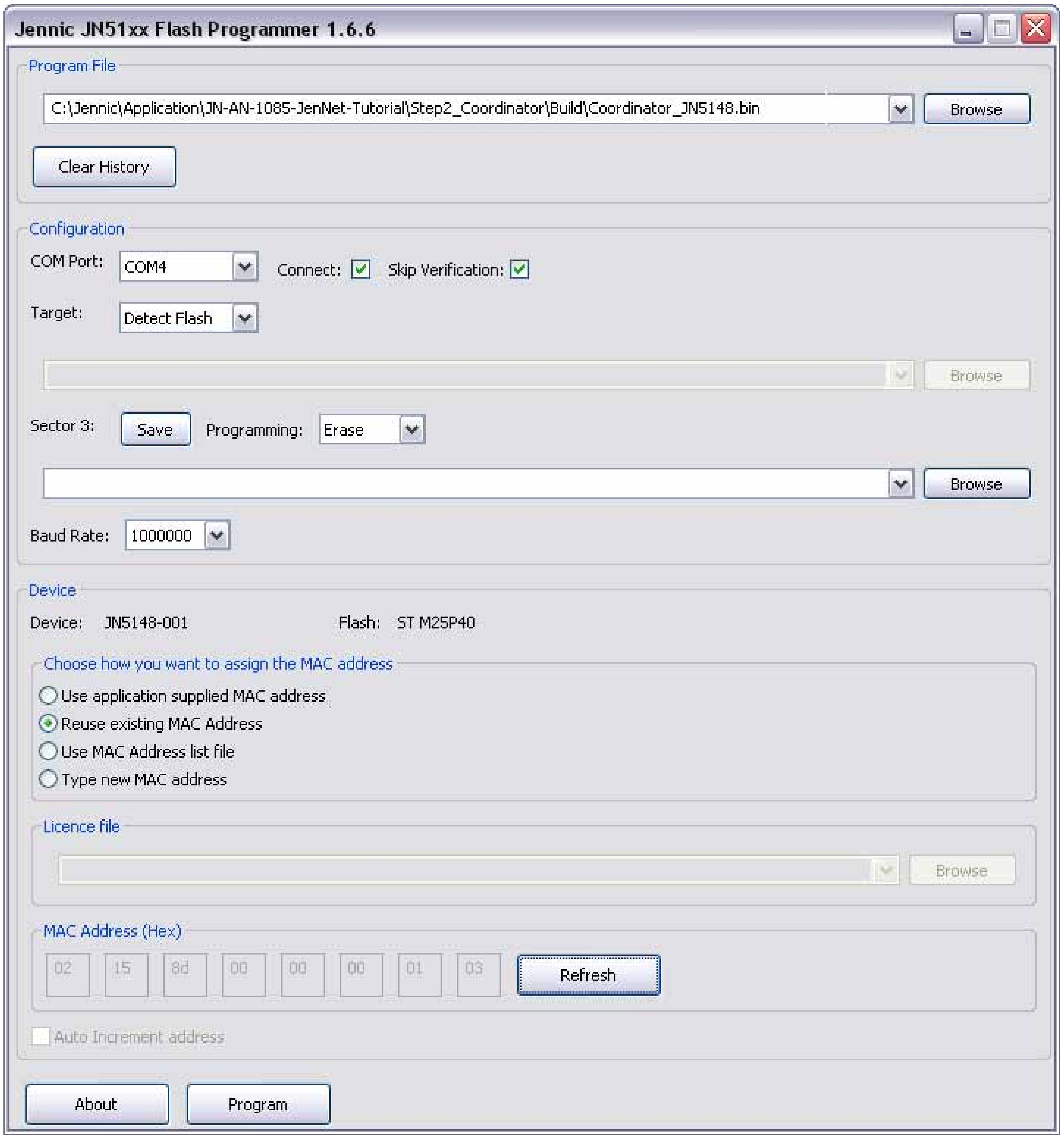
If compiling using Eclipse for JN5148 devices, select the **Step2_Coordinator** target from the 'build target' dropdown list . Ensure that the **Console** tab contains no errors.

If compiling using Code::Blocks for JN5139 devices, ensure the **Step2_Coordinator** project is active and press the 'build' button . Ensure that the **Build log** tab contains no errors.

To download the application to a board, the USB-to-serial FTDI cable supplied with the kit should be used to connect the PC to the board – a USB port of the PC must be connected to the board's UART0 connector, with the black wire connected to pin 1. The first time the USB-to-serial cable is connected to the PC, drivers may need to be installed.

The following procedure should be used to program the binary application into the board:

1. Put the board into programming mode, as follows: ensure the board is powered on, press and hold down the PROGRAM button, then press and release the RESET button, and finally release the PROGRAM button.
2. Start the JN51xx Flash Programmer from Eclipse by following the menu path **Run > External Tools > Flash GUI Tool** or from Code::Blocks by following the menu path **Tools > Flash GUI**.
3. Use the **Browse** button in the Flash programmer to select the binary file to be downloaded – **Coordinator_JN5148.bin** or **Coordinator_JN5139.bin**, located in the **Step2_Coordinator\Build** folder.
4. Ensure that the correct COM port is selected and that the **Connect** checkbox is ticked.
5. Ensure that the MAC address is non-zero – if it is zero, try putting the board into programming mode again and pressing the **Refresh** button.
6. Press the **Program** button to program the software into the board - a progress bar will be displayed while the binary is downloaded.
7. Close the application or uncheck the **Connect** checkbox when the download is complete, in order to free up the PC's COM Port to receive the **vPrintf()** output.


Jennic JN51xx Flash Programmer 1.6.6

Program File

C:\Jennic\Application\JN-AN-1085-JenNet-Tutorial\Step2_Coordinator\Build\Coordinator_JN5148.bin

Configuration

COM Port: COM4 Connect: ✓ Skip Verification: ✓

Target: Detect Flash

Sector 3: Save Programming: Erase

Baud Rate: 1000000

Device

Device: JN5148-001 Flash: ST M25P40

Choose how you want to assign the MAC address

○ Use application supplied MAC address
● Reuse existing MAC Address
○ Use MAC Address list file
○ Type new MAC address

Licence file

MAC Address (Hex)

02 15 8d 00 00 00 01 03

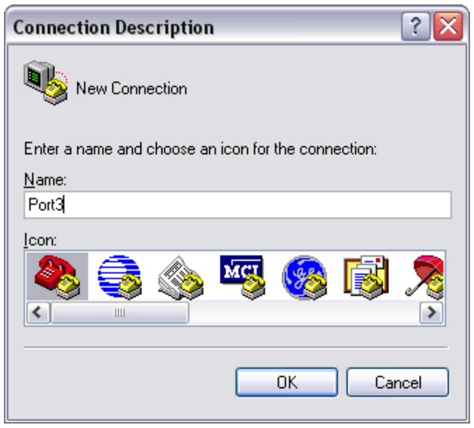
☐ Auto Increment address

Step 2.3 – Configuring a Terminal Application

To view the diagnostic information output by the **vPrintf()** functions in the code, it is necessary to run a terminal application connected to UART0. The terminal application should be configured to 19200 baud rate, 8 data bits, no parity, one stop bit, no flow control.

Instructions are provided below for those using Windows HyperTerminal on their PC:

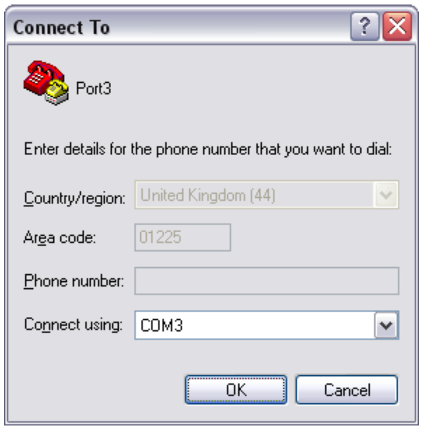
1. Open HyperTerminal: **Start > All Programs > Accessories > Communications > HyperTerminal**.
2. Access the New Connection dialogue box: **File > New Connection**. Type a name for the connection, then click **OK**.



Enter a name and choose an icon for the connection:

Name: Port3

3. Choose the serial communications port to which the board is connected and then click **OK**.



Enter details for the phone number that you want to dial:

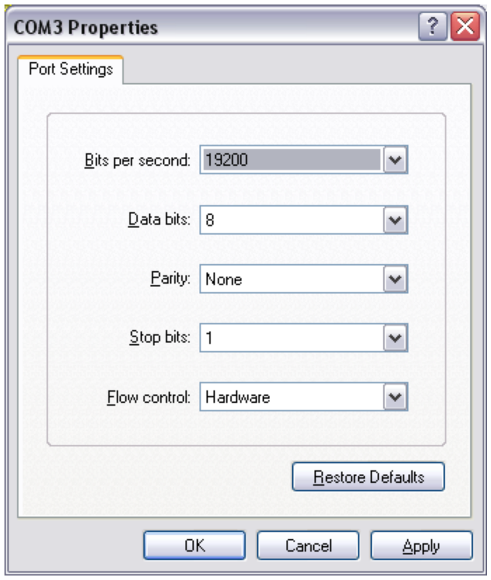
Country/region: United Kingdom (44)

Area code: 01225

Phone number:

Connect using: COM3

4. Set the port settings to 19200 bits per second, 8 data bits, no parity, 1 stop bit and no flow control, then click **OK**.



Bits per second: 19200

Data bits: 8

Parity: None

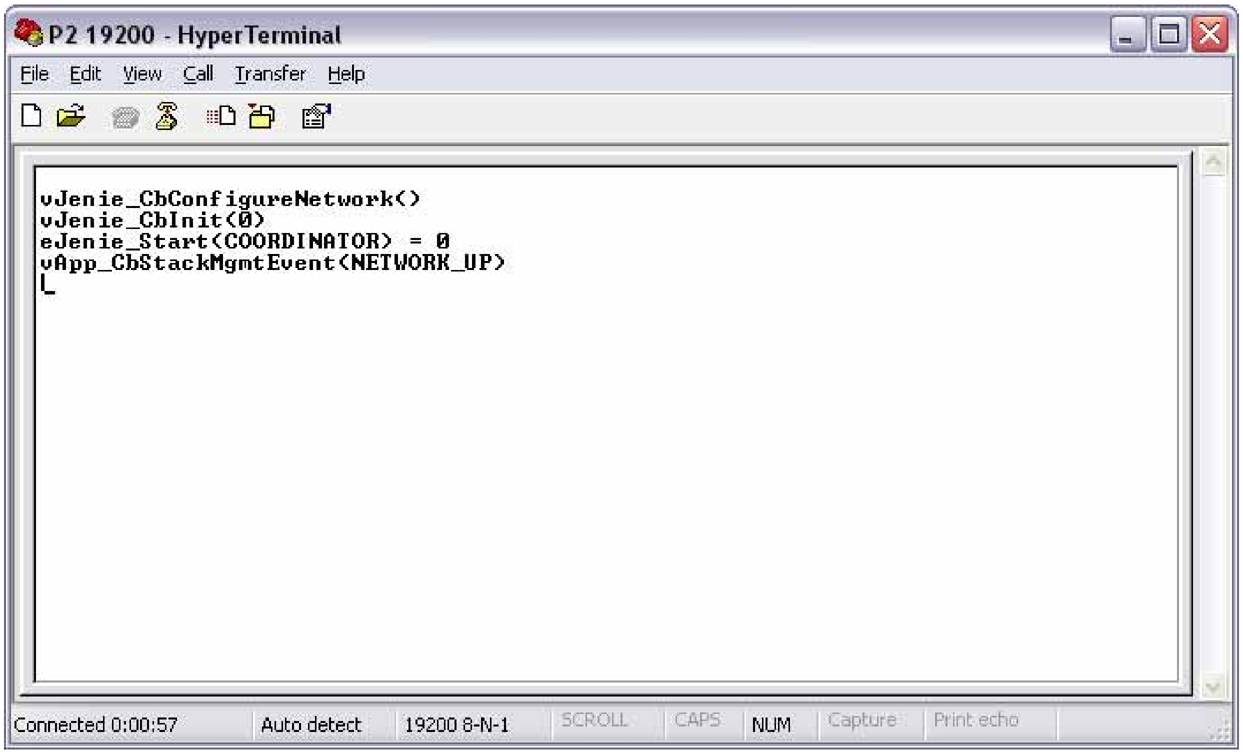
Stop bits: 1

Flow control: Hardware

Step 2.3 – Running the Application

After the download has completed, the board will remain in programming mode. Pressing and releasing the RESET button will reset the device and run the program.

LED 0 should flash quickly while the application waits for the network to start and should then extinguish. If viewing the diagnostics in a terminal application, the following sequence should be seen as the various stack events occur and function calls are made:



```
vJenie_CbConfigureNetwork()
vJenie_CbInit(0)
eJenie_Start(COORDINATOR) = 0
vApp_CbStackMgmtEvent(NETWORK_UP)
```

Before proceeding to the next step, the terminal application should be closed or the COM port it is using should be released to allow the port to be used to program the boards in the next step.

Note: If using HyperTerminal, selecting **Call > Disconnect** will free the COM port, while **Call > Call** will reconnect to the COM port. HyperTerminal will also have saved the settings to a file which can re-opened in later sessions.

Step 3 – Joining a Router to the Network

Router nodes always join an existing network. They are able to extend the network by allowing other nodes to join the network as their children. Routers need to be permanently powered, as they are used to route messages between nodes of the network.

This step adds a Router to the network. Stack management events are used on the Router and Co-ordinator nodes to determine when the network has been joined. A push-button is used to control when devices are allowed to join the network. A flashing LED and UART0 output are used to provide feedback. Once the network comprises more than one device, a push-button is used to send data between the devices, with the LEDs and UART0 providing feedback.

The new files for the Router for this step are located under the **Step3_Router** folder, the updated Co-ordinator files are located under the **Step3_Coordinator** folder, and the updated files shared by the Co-ordinator and Router devices are in the **Step3_Common** folder.

Step 3.1 – Code Changes

Router.c

Router.c is almost identical to **Coordinator.c** in that it contains the same set of Jenie callback functions which call the equivalent functions in **App.c**, allowing the **App.c** code to be used by both the Co-ordinator and Router.

The only difference between the two files is the call to **eJenie_Start()** in **eJenie_CbInit()**, which in **Router.c** specifies that the device should be started as a Router and attempt to join an existing network, rather than create a new network.

```
/* Start Jenie */
eStatus = eJenie_Start(E_JENIE_ROUTER);
/* Output function call to UART */
vPrintf("eJenie_Start(ROUTER) = %d\n", eStatus);
```

App.c

This section describes how the **App.c** file for Step 2 has been modified to produce the **App.c** file for Step 3.

Include Files

The following header file has been added:

```
#include <Button.h>          /* Button Interface {v3} */
```

We will be using functions from this file to read button presses.

Local Defines

The following local define has been added:

```
#define BUTTON_P_MASK (BUTTON_3_MASK << 1)  /* Mask for program button {v3} */
```

We will read the state of the PROGRAM button, which works differently from the other buttons. The above define will be used to track the state of the PROGRAM button.

Global Variables

The following global variables have been added:

```
PRIVATE uint8  u8Button;      /* Button state {v3} */
PRIVATE uint64 u64Parent;     /* Parent address {v3} */
PRIVATE uint64 u64Last;       /* Last address to send to us {v3} */
```

`u8Button` will be used as a bit mask to track the current state of the buttons.

`u64Parent` and **`u64Last`** will be used to store the addresses of the device's parent node and the last device to send data.

vApp_CbInit

A call to initialise the button hardware has been added:

```
/* Initialise LEDs and buttons {v3} */
vLedInitRfd();
vButtonInitRfd();
```

vApp_CbMain

Jenie provides a pair of functions to set and get the "permit joining" flag on a node. While this flag is set, the node allows other devices to join the network as the node's children. Later, we will toggle the flag whenever the PROGRAM button is released. To indicate whether "permit joining" is set, we read the current setting here and set LED 0 to flash if the flag is set.

```
/* Network running and permit join is on {v3} ? */
else if (bJenie_GetPermitJoin())
{
        /* Flash LED0 quickish while we are allowing joining */
        au8Led[0] = 0x04;
}
```

vApp_CbStackMgmtEvent

This function adds quite a lot of extra code. The first addition simply adds a variable to hold the return value from various Jenie functions.

```
teJenieStatusCode eStatus; /* Jenie status code {v3} */
```

Code then follows for each event in turn:

`E_JENIE_NETWORK_UP` is an event generated on the Co-ordinator when the network is started and on a Router when the network is joined. Additional code extracts the parent address from the event-specific data and stores it in **`u64Parent`** for later use. The additional event-specific data is added to the **vPrintf()** output. “Permit joining” is also enabled to allow other devices to join the network.

```
/* Indicates stack is up and running */
case E_JENIE_NETWORK_UP:
{
        /* Get pointer to correct primitive structure {v3} */
        tsNwkStartUp *psNwkStartUp = (tsNwkStartUp *) pvEventPrim;
        /* Output to UART */
        vPrintf("vApp_CbStackMgmtEvent(NETWORK_UP, %x:%x, %x:%x, %d, %x, %d)\n",
                (uint32)(psNwkStartUp->u64ParentAddress >> 32),
                (uint32)(psNwkStartUp->u64ParentAddress &  0xFFFFFFFF),
                (uint32)(psNwkStartUp->u64LocalAddress  >> 32),
                (uint32)(psNwkStartUp->u64LocalAddress  &  0xFFFFFFFF),
                psNwkStartUp->u16Depth,
                psNwkStartUp->u16PanID,
                psNwkStartUp->u8Channel);

        /* Network is now up */
        bNetworkUp = TRUE;
        /* Note our parent address {v3} */
        u64Parent = psNwkStartUp->u64ParentAddress;

        /* Turn on permit joining {v3} */
        eStatus = eJenie_SetPermitJoin(TRUE);
        /* Output to UART */
        vPrintf("eJenie_SetPermitJoin(%d) = %d\n",
                bJenie_GetPermitJoin(),
                eStatus);
}       break;
```

`E_JENIE_STACK_RESET` is an event generated when the network is lost. New code clears the parent address and disables “permit joining”.

```
/* Indicates stack has reset */
case E_JENIE_STACK_RESET:
{
        /* Output to UART */
        vPrintf("vApp_CbStackMgmtEvent(STACK_RESET)\n");

        /* Network is now down */
        bNetworkUp = FALSE;
        /* Clear our parent address {v3} */
        u64Parent = 0ULL;

        /* Turn off permit joining {v3} */
        eStatus = eJenie_SetPermitJoin(FALSE);
        /* Output to UART */
        vPrintf("eJenie_SetPermitJoin(%d) = %d\n",
                bJenie_GetPermitJoin(),
                eStatus);
}       break;
```

`E_JENIE_CHILD_JOINED` is an event generated when a new node joins the network as a child device. The address of the new child is extracted from the event-specific information and stored in **`u64Last`** as the address of a last device to contact us. Finally, “permit joining” is disabled to prevent further devices from joining the network.

```
/* Indicates child has joined {v3} */
case E_JENIE_CHILD_JOINED:
{
        /* Get pointer to correct primitive structure */
        tsChildJoined *psChildJoined = (tsChildJoined *) pvEventPrim;
        /* Output to UART */
        vPrintf("vApp_CbStackMgmtEvent(CHILD_JOINED, %x:%x)\n",
                (uint32)(psChildJoined->u64SrcAddress >> 32),
                (uint32)(psChildJoined->u64SrcAddress &  0xFFFFFFFF));

        /* Note our latest child */
        u64Last = psChildJoined->u64SrcAddress;

        /* Turn off permit joining */
        eStatus = eJenie_SetPermitJoin(FALSE);
        /* Output to UART */
        vPrintf("eJenie_SetPermitJoin(%d) = %d\n",
                bJenie_GetPermitJoin(),
                eStatus);
}       break;
```

`E_JENIE_CHILD_LEAVE` is an event generated when a child device leaves the network.

This event will be generated when the parent node fails to send data to the child node a number of times. The number of failures can be configured by setting the **`gJenie_MaxFailedPkts`** global variable in the **`vJenie_CbConfigureNetwork`** function. In this application, we have not set this variable, resulting in the default value of 5 being used.

In applications where data exchanges are infrequent, the loss of a child node may not be detected for a long period of time (during which data packets are not being sent from parent to child). In this case, it is possible to configure a Router child to automatically send a "ping" message to its parent to inform the parent that the child is still in the network. Setting **`gJenie_RouterPingPeriod`** in the **`vJenie_CbConfigureNetwork`** function configures the ping period. In this application, we have not set this variable, resulting in a default value of 5 seconds being used.

If a parent node does not receive any data from a Router child during a period equal to **`gJenie_RouterPingPeriod`** × **`gJenie_MaxFailedPkts`** then the parent node will assume the child is lost and generate the **`E_JENIE_CHILD_LEAVE`** event. With the default values used by this application, **`E_JENIE_CHILD_LEAVE`** will be generated after 25 seconds of inactivity from a Router child.

The address of the lost child is extracted from the event-specific information and if this address is the same as in **`u64Last`** then **`u64Last`** is cleared. Finally, "permit joining" is enabled to allow another device to join the network.

```
/* Indicates child has left {v3} */
case E_JENIE_CHILD_LEAVE:
{
        /* Get pointer to correct primitive structure */
        tsChildLeave *psChildLeave = (tsChildLeave *) pvEventPrim;
        /* Output to UART */
        vPrintf("vApp_CbStackMgmtEvent(CHILD_LEAVE, %x:%x)\n",
                (uint32)(psChildLeave->u64SrcAddress >> 32),
                (uint32)(psChildLeave->u64SrcAddress &  0xFFFFFFFF));

        /* Was that the last device to send us something ? */
        if (u64Last == psChildLeave->u64SrcAddress)
        {
                /* Clear the last address */
                u64Last = 0ULL;
        }

        /* Turn on permit joining */
        eStatus = eJenie_SetPermitJoin(TRUE);
        /* Output to UART */
        vPrintf("eJenie_SetPermitJoin(%d) = %d\n",
                bJenie_GetPermitJoin(),
                eStatus);
}       break;
```

vApp_CbStackDataEvent

This function handles events generated when data is received by the node:

`E_JENIE_DATA` is an event that occurs when data is received. The length of the message and message data are first validated. If the message is valid, the address of the node that sent the data is retained in **`u64Last`** and LED 0 is toggled between ON and OFF, if it is not flashing. The event-specific information, including the received data, is output to UART0.

`E_JENIE_DATA_ACK` is an event that occurs when an acknowledgement is received relating to data previously sent out. LED 0 is extinguished when this occurs.

```
PUBLIC void vApp_CbStackDataEvent(teEventType eEventType, void *pvEventPrim)
{
        /* Which event occurred ? */
        switch(eEventType)
        {
                /* Incoming data {v3} */
                case E_JENIE_DATA:
                {
                        /* Get pointer to correct primitive structure */
                        tsData *psData = (tsData *) pvEventPrim;
                        /* Output to UART */
                        vPrintf("vApp_CbStackDataEvent(DATA, %x:%x, %x, %d, '%s')\n",
                                (uint32)(psData->u64SrcAddress >> 32),
                                (uint32)(psData->u64SrcAddress &  0xFFFFFFFF),
                                 psData->u8MsgFlags,
                                 psData->u16Length,
                                (psData->pau8Data[psData->u16Length-1] == 0) ?
                                        (char *) psData->pau8Data : "");
                        /* Is this a button 0 message ? */
                        if (psData->u16Length   ==  3  &&
                            psData->pau8Data[0] == 'B' &&
                            psData->pau8Data[1] == '0')
                        {
                                /* Remember who sent this data to us */
                                u64Last = psData->u64SrcAddress;
                                /* Toggle LED0 */
                                if      (au8Led[0] == 0)    au8Led[0] = 0xFF;
                                else if (au8Led[0] == 0xFF) au8Led[0] = 0;
                        }
                }       break;

                /* Incoming data ack {v3} */
                case E_JENIE_DATA_ACK:
                {
                        /* Get pointer to correct primitive structure */
                        tsDataAck *psDataAck = (tsDataAck *) pvEventPrim;
                        /* Output to UART */
                        vPrintf("vApp_CbStackDataEvent(DATA_ACK, %x:%x)\n",
                                (uint32)(psDataAck->u64SrcAddress >> 32),
                                (uint32)(psDataAck->u64SrcAddress &  0xFFFFFFFF));
                        /* Turn off LED0 */
                        au8Led[0] = 0;
                }       break;

                default:
                {
                        // Unknown data event type
                }       break;
        }
}
```

vApp_CbHwEvent

In this function, a number of additional local variables are first declared:

```
uint8                  u8ButtonRead;    /* New button reading {v3} */
uint64                 u64Address;      /* Address to send data to {v3} */
teJenieStatusCode      eStatus;         /* Jenie status code {v3} */
```

The function then includes additional code to read and act upon changes to the buttons, each time our timer is activated, while the network is running.

The states of the standard buttons (0 and 1) are first read into a local variable. Next, the PROGRAM button is read. This button is attached to the SPI bus – therefore, if the SPI bus is in use, our reading will not be valid. In this case, the previous reading is preserved, otherwise the current state of the PROGRAM button is read.

Next, the program checks whether any buttons have changed state since the last time they were checked:

- If the PROGRAM button has been released, the “permit joining” flag is toggled.
- If button 0 has been released, a simple two-character message is built. The first byte is set to ‘B’, indicating that the message has been generated as the result of a button press. The second byte is set to ‘0’, to indicate that button 0 was pressed. An address is then selected to send the data message to - if the message is allowed to be sent, LED 0 is illuminated to indicate that data is to be sent. The **`TXOPTION_ACKREQ`** option is used to request that an acknowledgement is returned from the destination node.
- Finally, the new states of the buttons are stored for the next call.

```
/* Is the network up {v3} ? */
if (bNetworkUp)
{
        /* Read standard buttons */
        u8ButtonRead = u8ButtonReadRfd();
        /* If the SPI bus is in use –
           reuse the last value from the program button */
        if      (bJPI_SpiPollBusy()) u8ButtonRead |= (u8Button & BUTTON_P_MASK);
        /* SPI bus not in use and program button is pressed –
           set mask for program button UNDOCUMENTED */
        else if ((u8JPI_PowerStatus() & 0x10) == 0) u8ButtonRead |= BUTTON_P_MASK;
        /* Have the buttons changed ? */
        if (u8ButtonRead != u8Button)
        {
                /* Has the program button been released ? */
                if ((u8ButtonRead & BUTTON_P_MASK) == 0 &&
                        (u8Button & BUTTON_P_MASK) != 0)
                {
                        /* Toggle permit join setting */
                        eStatus = eJenie_SetPermitJoin(! bJenie_GetPermitJoin());
                        /* Output to UART */
                        vPrintf("eJenie_SetPermitJoin(%d) = %d\n",
                                bJenie_GetPermitJoin(),
                                eStatus);
                }

                /* Has button 0 been released ? */
                if ((u8ButtonRead & BUTTON_0_MASK) == 0 &&
                        (u8Button & BUTTON_0_MASK) != 0)
                {
                        /* Initialise data for transmission */
                        uint8 au8Data[3] = "B0";
                        /* Choose an address to send data to */
                        if (u64Last != 0ULL) u64Address = u64Last;
                        else                 u64Address = u64Parent;

                        /* Try to send data */
                        eStatus = eJenie_SendData(
                                u64Address,
                                au8Data,
                                3, TXOPTION_ACKREQ);
                        /* Output to UART */
                        vPrintf("eJenie_SendData(%x:%x, '%s', %d, %x) = %d\n",
                                (uint32)(u64Address >> 32),
                                (uint32)(u64Address &  0xFFFFFFFF),
                                au8Data,
                                3, TXOPTION_ACKREQ,
                                eStatus);

                        /* No errors ? */
                        if (E_JENIE_DEFERRED == eStatus)
                        {
                                /* Light LED0 */
                                au8Led[0] = 0xFF;
                        }
                }

                /* Note the current button reading */
                u8Button = u8ButtonRead;
        }
}
```

Step 3.2 – Building and Downloading the Application

This time, we need to build more than one project.

In Eclipse, build the **Step3_Coordinator** and **Step3_Router** build targets, one at a time.

In Code::Blocks, activate and build the **Step3_Coordinator** and **Step3_Router** projects, one at a time.

The binary files will be created in the **Step3_Coordinator\Build** and **Step3_Router\Build** folders, with the file names indicating the device type and chip type for which they were built. Use the Flash programmer to download the Co-ordinator binary to one board and the Router binary to a second board.

Note: It may be easier to use two cables and select different COM ports to download the two binaries.

Remember to close the Flash programmer or uncheck the **Connect** checkbox before continuing to run the application.

Step 3.3 – Running the Application

Once again, using a terminal application will provide feedback about the stack events and function calls that are being executed. As two nodes are being used, it is worth opening a second terminal window connected to an additional COM port. This will allow the output from both the Co-ordinator and Router nodes to be viewed at the same time.

1. Reset the Co-ordinator board. LED 0 will initially flash quickly while the network is created. Once the network is up and running, LED 0 will continue to flash slowly while "permit joining" is enabled. Pressing and releasing the PROGRAM button should toggle the "permit joining" setting and the LED should react accordingly. Finally, ensure that "permit joining" is enabled.
2. Reset the Router board. LED 0 will initially flash quickly while the Router attempts to join a network. Once it has joined a network, LED 0 on the Co-ordinator will be extinguished ("permit joining" is automatically disabled whenever a new child joins). LED 0 on the Router will flash more slowly as, in turn, "permit joining" is enabled on the Router. To disable "permit joining" on the Router, press and release the PROGRAM button - LED 0 should stop flashing.
3. Pressing and releasing button 0 on the Router should cause it to send a data message to its parent. LED 0 will be illuminated when the message is sent and extinguished again when the acknowledgement is received. On the Co-ordinator, LED 0 should toggle on and off each time button 0 is released on the Router.
4. Pressing and releasing button 0 on the Co-ordinator should work in the same way, toggling LED 0 on the Router.
5. Switching off the Co-ordinator node will result in the **`E_JENIE_STACK_RESET`** event being generated on the Router after 25 seconds, when 5 ping messages fail to be sent (or earlier, if button 0 is used on the Router node to try and send messages to the Co-ordinator). LED 0 will flash quickly on the Router as it automatically tries to re-join the network. Switching the Co-ordinator back on will allow the Router to re-join the network and operate as before.
6. Switching off the Router node will have a similar effect, resulting in the event **`E_JENIE_CHILD_LEAVE`** being generated on the Co-ordinator node. When this event occurs, the Co-ordinator will enable "permit joining" once again and LED 0 will flash to indicate this. Switching the Router node back on should allow it to re-join the network and operate as before.

Finally, the Router application can be programmed into additional boards. Use of the PROGRAM button will allow experimentation with formation of the network.

Button 0 on the Co-ordinator and Router will allow experimentation in sending data around the network. This button sends a message to another node with the following priority:

1. The last node to send to the local node
2. The last child node to join the local node
3. The parent node

Step 4 – Using Services

This step introduces JenNet services. A 'service' is a node feature or capability which can act as a data source or a data destination in network messaging.

Services are typically used as a convenient way for a node to indicate to the rest of the network that it is capable of receiving certain types of message, allowing other nodes in the network to find this node and bind to it. Once source and destination services are bound, there is no need to specify a destination address when sending data from the source to the destination(s).

A typical use of services is as follows:

1. The services of a node are registered with the network. The supported services are specified in a bit mask, where each service is represented by a particular bit.
2. A node that wishes to send messages first issues a request to the network for the destination services of interest. Any nodes that have registered the requested services will automatically respond, resulting in a series of service request responses being received by the requesting node. The data in the responses can then be used to bind a source service of the requesting node to one or more compatible destination services of responding nodes.
3. From this point, when the local node issues a message from its source service, the message will be sent to all bound destination services (on remote nodes).

This step of the tutorial will use prolonged button presses to register, request and send data to services.

The files required for this step are located in the **Step4_Common**, **Step4_Coordinator** and **Step4_Router** folders.

Step 4.1 – Code Changes

App.h

This section describes how the **App.h** file for Step 3 has been modified to produce the **App.h** file for Step 4.

Public Defines

Two new defines have been added to **App.h**, to specify the service number and service mask for the service that we are going to use. The service number is used in functions where we need to specify an individual service. The service mask is used in functions that operate on a set of services. In the mask, each bit corresponds to a single service, so the mask for service 1 is 0x1, while the mask for service 32 is 0x80000000.

```
#define SERVICE_NUMBER              1
#define SERVICE_MASK                (0x1 << (SERVICE_NUMBER-1))
```

App.c

This section describes how the **App.c** file for Step 3 has been modified to produce the **App.c** file for Step 4.

Local Defines

The following define has been added - this is a timeout used to time a long press of a button and is set to 2 seconds.

```
#define TIMEOUT_LONG_PRESS   200     /* Long button press timeout (in 10ms) {v4} */
```

Global Variables

The following global variables have been added:

```
PRIVATE uint16  au16ButtonTimer[2];  /* Timer for button presses {v4} */
PRIVATE uint32   u32Reg;             /* Registered services {v4} */
PRIVATE bool_t   bReg;               /* Successfully registered services {v4} */
PRIVATE bool_t   bBound;             /* Bound services {v4} */
PRIVATE bool_t   bSentService;       /* Data sent to service {v4} */
```

- Since we are going to use a long press of the buttons to register and request a service, **`au16ButtonTimer`** will be used to time how long each button is held down for.
- **`u32Reg`** will be used to store the bit mask of services that are already or currently being registered.
- **`bReg`** will be used to track if we currently have services registered.
- **`bBound`** will be used to track if we have bound our source service to a destination service.
- **`bSentService`** will be used to track if a message sent to a service is pending.

vApp_CbInit

The button timers are initialised to 0.

```
/* Reset button timers {v4} */
au16ButtonTimer[0] = 0;
au16ButtonTimer[1] = 0;
```

vApp_CbMain

If the network is running, “permit joining” is disabled and we have services registered, we will flash LED 0 slowly.

```
/* Network up and permit join off and services registered {v4} */
else if (bReg)
{
       /* Flash LED0 slowish while we have the service registered */
       au8Led[0] = 0x08;
}
```

vApp_CbStackMgmtEvent

When an End Device node registers services, they are actually registered in the parent node. When this registration is successful, the **E_JENIE_REG_SVC_RSP** event is generated. We will add End Device support to the application in the next step but, since we are introducing services in this step, we will add the code for this event now. The event is used to update the **bReg** global variable, depending on whether we currently have services registered.

```
/* Service registered response {v4} */
case E_JENIE_REG_SVC_RSP:
{
        /* Output to UART */
        vPrintf("vApp_CbStackMgmtEvent(REG_SVC_RSP)\n");

        /* Note if we have services registered or not */
        if (u32Reg == 0) bReg = FALSE;
        else             bReg = TRUE;
}       break;
```

When services are requested by a node, any remote nodes that have registered the requested services will automatically send a response which will generate an **E_JENIE_SVC_REQ_RSP** event in the requesting node.

The event structure provides the address of the responding node and the services supported by the node. If the supported services match those requested then we attempt to bind our source service to the responding node's destination service. If successful, we note that we have bound services and extinguish LED 0.

```
/* Service request response {v4} */
case E_JENIE_SVC_REQ_RSP:
{
        /* Get pointer to correct primitive structure */
        tsSvcReqRsp *psSvcReqRsp = (tsSvcReqRsp *) pvEventPrim;
        /* Output to UART */
        vPrintf("vApp_CbStackMgmtEvent(SVC_REQ_RSP, %x:%x, %x)\n",
                (uint32)(psSvcReqRsp->u64SrcAddress >> 32),
                (uint32)(psSvcReqRsp->u64SrcAddress &  0xFFFFFFFF),
                psSvcReqRsp->u32Services);

        /* Does this device support the service we requested ? */
        if (psSvcReqRsp->u32Services & SERVICE_MASK)
        {
                /* Try to bind the service */
                eStatus = eJenie_BindService(SERVICE_NUMBER,
                        psSvcReqRsp->u64SrcAddress, SERVICE_NUMBER);
                /* Output to UART */
                vPrintf("eJenie_BindService(%d, %x:%x, %d) = %d\n",
                        SERVICE_NUMBER,
                        (uint32)(psSvcReqRsp->u64SrcAddress >> 32),
                        (uint32)(psSvcReqRsp->u64SrcAddress &  0xFFFFFFFF),
                        SERVICE_NUMBER,
                        eStatus);
                /* Can we bind to the service ? */
                if (E_JENIE_SUCCESS == eStatus)
                {
                        /* Note we have bound */
                        bBound = TRUE;
                        /* Turn off LED 1 */
                        au8Led[1] = 0;
                }
        }
}       break;
```

When data is successfully sent from a device, the **`E_JENIE_PACKET_SENT`** event is generated (this applies to data sent directly to a node and also to data sent from a source service). We could make use of acknowledgements for data sent using services, in a similar way to those used for the direct messages in Step 3. However, if we have bound from a single source service to many destination services, this will result in a larger number of acknowledgement messages being returned. For the purposes of this tutorial, we will use the **`E_JENIE_PACKET_SENT`** event to track the progress of sending data using services.

Whenever we send data using services, we will set the **`bSentService`** flag. Therefore, if this flag is set when we detect the **`E_JENIE_PACKET_SENT`** event, we know that the event relates to the service data we issued. If this is the case, we will extinguish LED 1 and reset the flag for the next send.

```
/* Packet sent {v4} */
case E_JENIE_PACKET_SENT:
{
        /* Output to UART */
        vPrintf("vApp_CbStackMgmtEvent(PACKET_SENT)\n");

        /* Have we just sent to a service ? */
        if (bSentService)
        {
                /* Turn off LED1 */
                au8Led[1] = 0;
                /* Send to service is no longer pending */
                bSentService = FALSE;
        }
}       break;
```

vApp_CbStackDataEvent

When data sent to a service is received by a node, the **`E_JENIE_DATA_TO_SERVICE`** event is generated. The event structure members include the destination service to which the data was sent, the source address, and the length and content of the data itself. The destination service in the message is checked first then the length of message and message data are validated. When a valid message is received, LED 1 is toggled on or off.

```
/* Incoming data to service {v4} */
case E_JENIE_DATA_TO_SERVICE:
{
        /* Get pointer to correct primitive structure */
        tsDataToService *psDataToService = (tsDataToService *) pvEventPrim;
        /* Output to UART */
        vPrintf("vApp_CbStackDataEvent("
                "DATA_TO_SERVICE, %x:%x, %d, %d, %x, %d, '%s')\n",
               (uint32)(psDataToService->u64SrcAddress >> 32),
               (uint32)(psDataToService->u64SrcAddress &  0xFFFFFFFF),
                psDataToService->u8SrcService,
                psDataToService->u8DestService,
                psDataToService->u8MsgFlags,
                psDataToService->u16Length,
               (psDataToService->pau8Data[psDataToService->u16Length-1] == 0) ?
                       (char *) psDataToService->pau8Data : "");

        /* Is this to our service ? */
        if (psDataToService->u8DestService == SERVICE_NUMBER)
        {
                /* Is this a button 1 message ? */
                if (psDataToService->u16Length   ==  3  &&
                    psDataToService->pau8Data[0] == 'B' &&
                    psDataToService->pau8Data[1] == '1')
                {
                        /* Toggle LED1 */
                        if      (au8Led[1] == 0)    au8Led[1] = 0xFF;
                        else if (au8Led[1] == 0xFF) au8Led[1] = 0;
                }
        }
}   break;
```

vApp_CbHwEvent

The first new piece of code here is used to time how long button 0 is held down. When the timer event occurs for the first timer, LED 0 is extinguished (if it has been left on by a previous LED toggling message).

The timer for button 0 is then incremented.

If button 0 has then been held down for a long time (two seconds) then the registered services mask is toggled between our service mask and no services, and then the updated service mask is registered. For a Co-ordinator or Router, this should return **`E_JENIE_SUCCESS`**, in which case the **`bReg`** flag is updated to track if we currently have services registered. On an End Device, we must wait for the **`E_JENIE_REG_SVC_RSP`** event described earlier. While the **`bReg`** flag is set, code in the function **vApp_CbMain()** causes LED 0 to flash slowly.

```
/* Is button 0 down {v4}? */
if (u8ButtonRead & BUTTON_0_MASK)
{
        /* Are we just starting the timer ? */
        if (au16ButtonTimer[0] == 0)
        {
                /* If LED0 is permamently on –
                   turn it off so we can see feedback on the LED */
                if (au8Led[0] == 0xFF) au8Led[0] = 0;
        }
        /* Increment the button timer */
        au16ButtonTimer[0]++;
        /* Has the button been down a long time? */
        if (au16ButtonTimer[0] == TIMEOUT_LONG_PRESS)
        {
                /* Toggle registered services */
                if (u32Reg == 0) u32Reg = SERVICE_MASK;
                else             u32Reg = 0;
                /* Try to register services */
                eStatus = eJenie_RegisterServices(u32Reg);
                /* Output to UART */
                vPrintf("eJenie_RegisterServices(%x) = %d\n", u32Reg, eStatus);

                /* Did they register immediately ? */
                if (E_JENIE_SUCCESS == eStatus)
                {
                        /* Note if we have services registered or not */
                        if (u32Reg == 0) bReg = FALSE;
                        else             bReg = TRUE;
                }
        }
}
```

Similar code is then used to time how long button 1 is held down. When the timer event occurs for the first timer, LED 1 is extinguished (if it has been left on by a previous LED toggling message).

The timer for button 1 is then incremented.

If button 1 has then been held down for a long time (two seconds) then a request is made for the services in which we are interested in binding. If this request does not fail, we will light LED 1 to indicate that the request is pending.

```
/* Is button 1 down {v4} ? */
if (u8ButtonRead & BUTTON_1_MASK)
{
        /* Are we just starting the timer ? */
        if (au16ButtonTimer[1] == 0)
        {
                /* If LED0 is permamently on –
                   turn it off so we can see feedback on the LED */
                if (au8Led[1] == 0xFF) au8Led[1] = 0;
        }
        /* Increment the button timer */
        au16ButtonTimer[1]++;
        /* Has the button been down a long time? */
        if (au16ButtonTimer[1] == TIMEOUT_LONG_PRESS)
        {
                /* Try to request the services */
                eStatus = eJenie_RequestServices(SERVICE_MASK, FALSE);
                /* Output to UART */
                vPrintf("eJenie_RequestServices(%x, FALSE) = %d\n",
                        SERVICE_MASK, eStatus);

                /* Error free ? */
                if (E_JENIE_DEFERRED == eStatus)
                {
                        /* Light LED1 */
                        au8Led[1] = 0xFF;
                }
        }
}
```

Whenever button 0 is held down for more a long time to register a service, we do not want to perform the data send that normally occurs when button 0 is released. An extra **`if`** statement around the ‘send data’ code protects against this happening. The timer for button 0 is also reset when the button is released.

```
/* Was the button not down a long time {v4} ? */
if (au16ButtonTimer[0] < TIMEOUT_LONG_PRESS)
{
        /* (Send data code here as before) */

        /* Reset the button timer {v4} */
        au16ButtonTimer[0] = 0;
}
```

Code is then added for the case of button 1 being released. If services have been bound, a data message is built and sent from the bound service, LED 1 is illuminated and we note that we have sent data to a service. We finish by resetting the button 1 timer to 0.

```
/* Has button 1 been released ? */
if ((u8ButtonRead & BUTTON_1_MASK) == 0 && (u8Button & BUTTON_1_MASK) != 0)
{
        /* Was the button not down a long time {v4} ? */
        if (au16ButtonTimer[1] < TIMEOUT_LONG_PRESS)
        {
                /* Have we bound services yet ? */
                if (bBound)
                {
                        /* Initialise data for transmission */
                        uint8 au8Data[8] = "B1";

                        /* Try to send data */
                        eStatus = eJenie_SendDataToBoundService(
                                SERVICE_NUMBER, au8Data, 3, 0);
                        /* Output to UART */
                        vPrintf("eJenie_SendDataToBoundService"
                                " (%d, '%s', %d, %x) = %d\n",
                                SERVICE_NUMBER,
                                au8Data,
                                3,
                                0,
                                eStatus);

                        /* Can we send data ? */
                        if (E_JENIE_DEFERRED == eStatus)
                        {
                                /* Light LED1 */
                                au8Led[1] = 0xFF;
                                /* Note we've sent data to a service */
                                bSentService = TRUE;
                        }
                }
        }
        /* Reset the button timer {v4}*/
        au16ButtonTimer[1] = 0;
}
```

Step 4.2 – Building, Downloading and Running the Application

Once again, we need to build more than one project.

1. Build and download the binaries for Step 4 to two boards.

> **Note:** It may be easier to use two cables and select different COM ports to download the two binaries.

2. Starting, joining and sending data directly in the network should operate as before, but ensure that “permit joining” is disabled on all devices before proceeding to experiment with services.

3. Press and hold down button 0 on a node for two seconds. This should register the node’s service for other nodes to discover and bind to. LED 0 should flash slowly while the service is registered. Holding down button 0 for another two seconds should unregister the service. If necessary, re-register the service so that the other node can discover and bind to it.

4. Press and hold down button 1 on a different node for two seconds. LED 1 should illuminate when the request is made, and extinguish when the request response is received and the service is bound. Pressing and releasing button 1 on its own should send data using the bound service, resulting in LED 1 toggling on and off at the receiving device.

> **Note:** A service being registered only affects whether other devices can discover that service by requesting services from the network. A previously bound service will continue to send data correctly even if the service is unregistered.

Programming the Router application into additional boards will allow experimentation with binding using one-to-many and many-to-one bindings.

Step 5 – Joining an End Device to the Network

This step adds an End Device to the network.

An End Device is able to sleep in order to conserve power when it does not need to be processing, transmitting or receiving. We will use the PROGRAM button to put the End Device to sleep, and allow buttons 0 and 1 to wake it from sleep.

- While the End Device is awake, a timer will be run that will allow the node to send a short message to its parent regularly, if no other messages are being sent. This ensures that the parent node knows the End Device is still a member of the network.
- While sleeping, the End Device will wake regularly, automatically pinging its parent node, then quickly going back to sleep. Again, this is to ensure that the parent knows the End Device is still a member of the network.

Since an End Device is not always active, it is not able to take on other devices as children, so does not support the "permit joining" feature.

The End Device application will otherwise operate in the same way as the other nodes.

Note: End Device nodes are required to use a different Jenie library file from the Co-ordinator and Router nodes, due to the differences in functionality between the node types. The type of node that a device can run as is therefore determined at compile time by the Jenie library that is linked into the application.

The Jenie library is set in the Eclipse makefile or Code::Blocks project file (accessed from the Linker tab of the Project Build Options window in Code::Blocks). The libraries are as follows:

Jenie_TreeCRLib.a for Co-ordinator and Router nodes
Jenie_TreeEDLib.a for End Device nodes

The files required for this step are located in the **Step5_Common**, **Step5_Coordinator**, **Step5_Router** and **Step5_EndDevice** folders.

Step 5.1 – Code Changes

App.h

This section describes how the **App.h** file for Step 4 has been modified to produce the **App.h** file for Step 5.

We will continue to share the code in **App.h** and **App.c** between all the node types, although there will now be small differences between the Co-ordinator/Router code and the End Device code, due to the different ways in which the nodes operate. We will add an extra parameter to the **vApp_CbInit()** function to indicate whether or not the node is an End Device. The prototype for **vApp_CbInit()** is therefore updated.

```
PUBLIC void vApp_CbInit(bool_t, bool_t);     /* Warm start, End device {v5} */
```

Coordinator.c and Router.c

This section describes how the files **Coordinator.c** and **Router.c** for Step 4 have been modified to produce the **Coordinator.c** and **Router.c** files for Step 5.

vJenie_CbConfigureNetwork

Both applications specify a timeout period using **`gJenie_EndDeviceChildActivityTimeout`** for their End Device children. If no messages are received from an End Device child for this period of time, the child node will be assumed to have left the network.

```
/* Set inactivity timer for end device children to 25 seconds {v5} */
gJenie_EndDeviceChildActivityTimeout = 250;
```

vJenie_CbInit

Both applications have the new parameter added to the function **vApp_CbInit()** to indicate that the host nodes are not End Devices.

```
/* Initialise application, not end device {v5} */
vApp_CbInit(bWarmStart, FALSE);
```

EndDevice.c

This new file **EndDevice.c** contains the End Device application, and is identical to **Coordinator.c** and **Router.c** with the following exceptions:

vJenie_CbConfigureNetwork

An additional global variable, **`gJenie_EndDevicePollPeriod`**, is set here. An End Device must poll its parent to receive data sent to the End Device. Setting this variable configures the rate at which the End Device will automatically poll its parent for new data – the polling interval is set in units of 100ms. For our application, we will poll once every half-second.

The global variable **`gJenie_EndDeviceScanSleep`** is also set here. If an End Device fails to find a network to join, it will sleep for a period before waking and trying to scan again. This variable specifies how long an End Device should sleep between scans. For our application, we will sleep for a second.

```
/* Poll every half a second (in 100ms intervals) */
gJenie_EndDevicePollPeriod  = 5;
/* Sleep for one second between scans (in 1ms intervals) */
gJenie_EndDeviceScanSleep   = 1000;
```

Note that the Routing table and **`gJenie_EndDeviceChildActivityTimeout`** global variables are not set here, as End Device nodes do not provide routing functionality or have child nodes.

vJenie_CbInit

This function call ensures that the application is initialised and JenNet is started for an End Device.

```
PUBLIC void vJenie_CbInit(bool_t bWarmStart)
{
        teJenieStatusCode eStatus; /* Jenie status code */

        /* Warm start - reopen UART for printf use */
        if (bWarmStart) vUART_printInit();
        /* Output function call to UART */
        vPrintf("vJenie_CbInit(%d)\n", bWarmStart);

        /* Initialise application, end device {v5} */
        vApp_CbInit(bWarmStart, TRUE);

        /* Start Jenie */
        eStatus = eJenie_Start(E_JENIE_END_DEVICE);
        /* Output function call to UART */
        vPrintf("eJenie_Start(END_DEVICE) = %d\n", eStatus);
}
```

App.c

Defines

The following defines are added for various timeouts and timers.

```
#define TIMEOUT_IDLE_ASLEEP   2      /* Idle timeout when sleeping (in 10ms) {v5} */
#define TIMEOUT_IDLE_AWAKE  500      /* Idle timeout when awake (in 10ms) {v5} */
#define PERIOD_SLEEP       5000      /* Sleep period (in ms) {v5} */
```

Global Variables

bEndDevice is used to track whether the application is running as an End Device.

bSleep is used to track if the node is in sleep mode.

u16IdleTimer is used to time how long the node is idle (not sending any data).

```
PRIVATE bool_t    bEndDevice;           /* Device is an end device {v5} */
PRIVATE bool_t    bSleep;               /* End device is in sleeping mode {v5} */
PRIVATE uint16    u16IdleTimer;         /* Timer for idleness {v5} */
```

vApp_CbInit

The new End Device parameter is passed into this function.

```
PUBLIC void vApp_CbInit(      bool_t bWarmStart,
                              bool_t bInitEndDevice) /* Initialising end device {v5} */
```

Some extra initialisation is added for all node types. This is largely to ensure that functionality specific to End Devices is not activated on Co-ordinator and Router nodes.

For a cold start, the **bSleep** flag is set to FALSE, as the End Device will only go into sleep mode once the PROGRAM button has been pressed.

```
/* We are not in sleep mode {v5} */
bSleep = FALSE;
```

The idle timer (which will count down) is initially switched off by setting it to zero.

```
/* Don't run idle timer {v5} */
u16IdleTimer = 0;
```

The End Device parameter flag is stored in the **bEndDevice** global variable.

```
/* Note if we are an end device {v5} */
bEndDevice = bInitEndDevice;
```

If the node is running as an End Device, we perform some extra initialisation.

Waking from sleep with the network active is indicated by illuminating both LEDs on the node.

If any of the buttons are down, it is assumed that the node woke as the result of a button press, so the node is taken out of sleep mode.

If the node is in sleep mode, the idle timer is set to a very low value (20 ms). This allows the End Device node enough time to ping its parent, when it wakes, before going to sleep again when the idle timer expires. The number of sleep cycles between automatic pings can be configured using the **`gJenie_EndDevicePingInterval`**. This application does not set this variable, resulting in the default value of 1 being used (meaning that the parent will be pinged every time the End Device wakes from sleep).

If the node is not in sleep mode, the idle timer is set to a higher value (5 seconds).

The inputs for the buttons are configured to wake the End Device from sleep when they are pressed down.

Finally, the length of time for which the End Device should sleep is set to the value defined earlier (5 seconds).

```
/* Is this an end device {v5} ? */
if (bEndDevice)
{
        /* Warm start and the network is up ? */
        if (bWarmStart && bNetworkUp)
        {
                /* Turn on LEDs indicating we've woken up */
                au8Led[0] = 0xFF;
                au8Led[1] = 0xFF;
                /* Have we woken because of a button press ? */
                if (u8ButtonReadRfd())
                {
                        /* Make sure we are not in sleep mode */
                        bSleep = FALSE;
                }
                /* Set idle timer dependent upon sleep mode */
                if (bSleep) u16IdleTimer = TIMEOUT_IDLE_ASLEEP;
                else        u16IdleTimer = TIMEOUT_IDLE_AWAKE;
        }
        /* Set interrupts on buttons to wake us up */
        vJPI_DioWake(BUTTON_ALL_MASK_RFD_PIN, 0, 0, BUTTON_ALL_MASK_RFD_PIN);
        /* Set sleep period */
        eJenie_SetSleepPeriod(PERIOD_SLEEP);
}
```

vApp_CbStackMgmtEvent

For the **`E_JENIE_NETWORK_UP`** event, the idle timer is started with a value which is dependent upon the node being in sleep mode.

Additionally, “permit joining” is only enabled if the node is not running as an End Device.

```
/* End device {v5} ? */
if (bEndDevice)
{
        /* Set idle timer dependent upon sleep mode */
        if (bSleep) u16IdleTimer = TIMEOUT_IDLE_ASLEEP;
        else        u16IdleTimer = TIMEOUT_IDLE_AWAKE;
}

/* Turn on permit joining {v3} (for coord and routers) {v5} */
eStatus = eJenie_SetPermitJoin(! bEndDevice);
```

For the **`E_JENIE_STACK_RESET`** event, we cancel the idle timer.

```
/* End device - stop idle timer running */
if (bEndDevice) u16IdleTimer = 0;
```

vApp_CbHwEvent

When the End Device wakes from sleep, both LEDs are illuminated to indicate wake from sleep. However, code in **vApp_CbHwEvent()** will extinguish one of those LEDs when the timer for the button begins. The code that extinguishes the LEDs will be adjusted to extinguish them just after the timer begins. For a waking End Device, this will provide a short period when both LEDs are on.

```
/* Are we just starting the timer –
   but allow it to be on a short time for a waking end device {v5} ? */
if (au16ButtonTimer[0] == 2)
```

We must take different actions when the PROGRAM button is released, depending on whether the node is running as an End Device:

- For an End Device, we set the sleep mode flag and set the idle timer to a very short period (20 ms). This will result in the device going to sleep when the timer expires.
- For a Co-ordinator and Router, “permit joining” will be toggled as before (code not shown).

```
/* End device {v5} ? */
if (bEndDevice)
{
        /* We are going into sleep mode */
        bSleep = TRUE;
        /* Start idle timer for entering sleep mode */
        u16IdleTimer = TIMEOUT_IDLE_ASLEEP;
}
```

When data is transmitted by an End Device as a result of one of the buttons being released, while not in sleep mode the idle timer is restarted, as the messages will be sent via the parent node.

```
/* End device and not in sleep mode - reset a long wake timer {v5} */
if (bEndDevice && ! bSleep) u16IdleTimer = TIMEOUT_IDLE_AWAKE;
```

Finally, if running, the idle timer is decremented.

If the timer expires while in sleep mode, the LEDs are extinguished and the node is put to sleep.

If the timer expires while not in sleep mode, a short message is sent to the parent node and the idle timer is restarted.

```
/* Idle timer running {v5} ? */
if (u16IdleTimer > 0)
{
        /* Decrement the idle timer */
        u16IdleTimer--;
        /* Idle timer expired and we are and end device and network is up ? */
        if (u16IdleTimer == 0 && bEndDevice && bNetworkUp)
        {
                /* Are we in sleep mode ? */
                if (bSleep)
                {
                        /* Turn off LEDs as we're going to sleep */
                        au8Led[0] = 0;
                        au8Led[1] = 0;
                        /* Output to UART (before sleeping) */
                        vPrintf("eJenie_Sleep(E_JENIE_SLEEP_OSCON_RAMON)\n");
                        /* Wait for tx fifo and shift register to be empty */
                        while ((u8JPI_UartReadLineStatus(E_JPI_UART_0) &
                                (E_JPI_UART_LS_THRE | E_JPI_UART_LS_TEMT)) !=
                                (E_JPI_UART_LS_THRE | E_JPI_UART_LS_TEMT));
                        /* Go to sleep holding memory */
                        (void) eJenie_Sleep(E_JENIE_SLEEP_OSCON_RAMON);
                }
                else
                {
                        /* Initialise data for transmission */
                        uint8 au8Data[2] = "I";
                        /* Send to our parent to let it know we are awake */
                        eStatus = eJenie_SendData(u64Parent, au8Data, 2, 0);
                        /* Output to UART */
                        vPrintf("eJenie_SendData(%x:%x, '%s', %d, %x) = %d\n",
                                (uint32)(u64Parent >> 32),
                                (uint32)(u64Parent &  0xFFFFFFFF),
                                au8Data,
                                2,
                                0,
                                eStatus);
                        /* Restart a long idle timer */
                        u16IdleTimer = TIMEOUT_IDLE_AWAKE;
                }
        }
}
```

Step 5.2 – Building, Downloading and Running the Application

Once again, we need to build more than one project:

1. Build and download the three Step 5 binaries, including the new End Device binary, to three evaluation kit boards.
2. The new End Device should operate in the same way as the other devices, except it will not allow other devices to join to it.
3. When the End Device is trying to join the network, you may notice that LED 0 stops flashing for a time, as the node will sleep between attempts.
4. When sending data and requesting services, you may notice a longer delay between the send and acknowledgement, since the End Device needs to poll its parent for data every half-second.
5. Pressing and releasing the PROGRAM button on the End Device should put the node to sleep. Pressing button 0 or 1 will wake it again. On releasing button 0 or 1 after the node has woken, the node should perform the task programmed for that button. When waking, any message sent to the End Device while it was asleep should be retrieved from its parent and acted upon (note that the parent has limited space for buffering messages).
6. While the node is asleep, both LEDs should flash every 5 seconds as the node briefly wakes to ping its parent and keep its place in the network.
7. The **`E_JENIE_STACK_RESET`** and **`E_JENIE_CHILD_LEAVE`** events should be generated in the same way as for Router and Co-ordinator nodes, when the parent and End Device nodes are powered off and on.

Compatibility

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

Product Type	**Part Number**	**Version**	**Supported Chips**
Evaluation Kits	JN5139-EK000	-	JN5139
	JN5139-EK010	-	JN5139
	JN5148-EK010	-	JN5148
SDK Libraries	JN-SW-4030	v1.5	JN5139
	JN-SW-4040	v1.4	JN5148
SDK Toolchain	JN-SW-4031	v1.1	JN5139
	JN-SW-4041	v1.1	JN5148

Revision History

Version	Notes
1.0	First release
1.1	Updated for Jenie v1.3. Corrected step numbers in project workspace files. Added diagnostic output to UART 0. Received data is extracted from the event structures. Simplified button used, prolonged presses of the buttons access the service features (rather than complex button sequences). Uses tick timer (automatically configured by Jenie), instead of timer 1. Explanations added regarding how E_JENIE_CHILD_LEAVE and E_JENIE_STACK_RESET events are generated. End Device sleeps for only one second between scans. End Device wakes regularly from sleep to ping parent. End Device sends data to parent when idle, to stay in network.
1.2	Updated to support the JN5148 device.
1.3	Updated for inclusion of Jenie in JN5148 SDK Libraries installer.
1.4	Terminology and logo changes made

Important Notice

NXP Laboratories UK Ltd
Furnival Street
Sheffield
S1 4QT
United Kingdom

Tel: +44 (0)114 281 2655
Fax: +44 (0)114 281 2951
E-mail: info@jennic.com

For the contact details of your local Jennic office or distributor, refer to the Jennic web site:

www.Jennic.com
TECHNOLOGY FOR A CHANGING WORLD