A brief tutorial on SimpliciTI 1.1.1

Loïc Lemaître

November 3, 2010

Abstract

This document presents a short introduction to the Texas Instrument SimpliciTI library V1.1.1 in the WSN430 platform context. It explains how to port it on the WSN430 hardware platform, to make it compilable with MSPGCC, and gives a very simple program example to help you to start with SimpliciTI on WSN430. More TI documentations describing SimpliciTI features are available on the official website : `http://www.ti.com/corp/docs/landing/simpliciTI/index.htm`

Contents

Chapter 1

Introduction

SimpliciTI is a TI proprietary low power Radio Frequency network protocol. It is intended to support development of wireless end user devices in environment in which the network support is simple and the developer desires a simple means to do messaging over air. Thus SimpliciTI supplies APIs in order to manage easily messaging between devices.

SimpliciTI is considered as a library and not as an operating system, since it does not implement task handler.

SimpliciTI main features are :

- low memory needs (<8kB flash and 1kB ram depending on the configuration);
- advance network control (security, radio frequency agility, ...);
- sleeping modes support.

Chapter 2

Architecture overview

2.1 Network topologies

It supports 2 basic topologies: a strictly peer-to-peer and a star topology in which the star hub is a peer to every other device .

2.2 Device types

SimpliciTI allows user to implement three device types : End Device, Range Extender, and Access Point. Note that a hardware device may host several SimpliciTI devices either of same type or of different type.

2.2.1 End Device

It is the base element of the network. It generally supports most of the sensors or actuators of the network. A strictly peer-to-peer network is exclusively composed of end devices (and eventually range extenders).

2.2.2 Access Point

It supports such features and functions as store-and-forward support for sleeping End Devices, management of network devices in terms of membership permissions, linking permissions, security keys, etc. The Access Point can also support End Device functionality.

In the star topology the Access Point acts as the hub of the network.

2.2.3 Range Extender

These devices are intended to repeat frames in order to extend the network range. Due to their function, they are always on. Networks are currently limited to 4 range extenders.

2.3 SimpliciTI layers

SimpliciTI is organized in 3 layers : Data Link/Physic, Network, and Application.

2.3.1 Application

This is the only layer that the developer needs to implement. It is where he develops his application (to manage sensors for example), and implements network communication, by using SimpliciTI network APIs or network applications.

Note that it is in this layer that the developer needs to implement reliable transport if required, as there is no Transport layer.

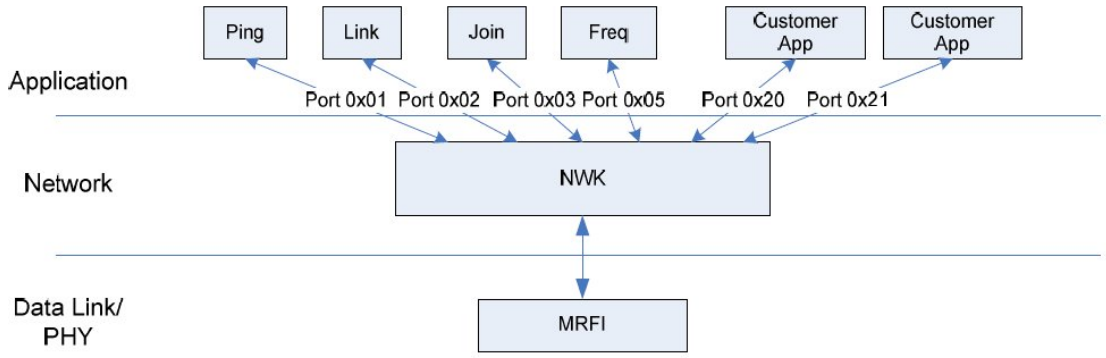

Figure 2.1: SimpliciTI logical layers

2.3.2 Network

This layer manages the Rx and Tx queues and dispatches frames to their destination. The destination is always an application designated by a Port number.

Network applications are internal peer-to-peer objects intended to manage network. They work on a predefined port and are not intended to be used by the developer (except `Ping` for debugging purposes) Their usage depends on the SimpliciTI device type. These applications are:

- `Ping` (Port 0x01): to detect the presence of a specific device.
- `Link` (Port 0x02): to support the link management of two peers.
- `Join` (Port 0x03): to guard entry to the network in topologies with APs.
- `Security` (Port 0x04): to change security information such as encryption keys and encryption context.
- `Freq` (Port 0x05): to perform change channel, change channel request or echo request.
- `Mgmt` (Port 0x06): general management port to be used to manage the device.

Source code files for network layer are located in `/Components/simpliciti`.

2.3.3 Data Link/Physic

This layer may be divided in 2 entities:

- BSP (Board Support Package): to abstract the SPI interface from the NWK layer calls that interact with the radio (`/Components/mrfi`);
- MRFI (Minimal RF Interface): to encapsulate the differences between supported hardware radios, toward the network layer (`/Components/bsp`).

Chapter 3

Hardware support

3.1 Radios

SimpliciTI supports 5 families of TI radios :

- Family 1 : CC1100, CC1101, CC2500
- Family 2 : CC2510, CC2511, CC1110, CC1111
- Family 3 : CC2520
- Family 4 : CC2430
- Family 5 : CC2530

Source code files for these 5 families of radio are located in the `/mrfi/radios` folder.

3.2 Microcontroller

SimpliciTI implements 2 families of microcontrollers : Intel 8051 and TI MSP430 (source code directory : `/bsp/mcus`).

3.3 Boards

The following boards are supported : CC2430DB, CC2530EM, EXP461x, EZ430RF, RFUSB, SRF04EB, SRF05EB (source code directory : `/bsp/boards`).

3.4 Devices

As a convenience SimpliciTI also supports LEDs and button/switch peripherals attached to GPIO pins of the microcontroller. But no other services are provided such as UART drivers, LCD drivers, or timer services (source code directory : `/bsp/drivers` directory).

Chapter 4

APIs supplied

APIs enable user to implement a reliable network with little effort. But we have to keep in mind that the resulting network sacrifices flexibility for simplicity.

Here are the different APIs supplied by SimpliciTI:

- Initialization
  - `smplStatus_t SMPL_Init(uint8 (*pCB)(linkID))`
- Linking (bi-directional by default)
  - `smplStatus_t SMPL _Link(linkID_t *linkID)`
  - `smplStatus_t SMPL _LinkListen(linkID_t *linkID)`
- Peer to peer messaging
  - `smplStatus_t SMPL _Send(linkID_t lid, uint8 *msg, uint8 len)`
  - `smplStatus_t SMPL _Receive(linkID_t lid, uint8 *msg, uint8 *len)`
- Configuration
  - `void SMPL_Ioctl(ioctlObject_t object, ioctlAction_t action, void *val)`

Chapter 5

Porting SimpliciTI on the WSN430 hardware platform

5.1 Do it yourself

If you want to be sure to use the latest available version of SimpliciTI, you can put the wsn430 board code of the old version SimpliciTI into the new one.

5.1.1 Adding WSN430 board files

1. Get the latest release of SimpliciTI from SimpliciTI page of the TI website[1]. Choose SimpliciTI for IAR link, download and install it.
2. Into the `/Components/bsp/board/` directory of SimpliciTI installed files, add the `WSN430` folder from `/Components/bsp/board/` directory of the `simpliciti-wsn430-v1.1.1.tar.gz` archive, available on SimpliciTI page of the Senstools website[2] (SimpliciTI ported for WSN430).
3. Into the `/Projects/Examples/` directory of SimpliciTI installed files, add the `WSN430` folder from `/Projects/Examples/` directory of the `simpliciti-wsn430-v1.1.1.tar.gz` archive, available on SimpliciTI page of the Senstools website[3] (SimpliciTI ported for WSN430).

5.1.2 Compiling SimpliciTI with MSPGCC

SimpliciTI is written to be compiled with the IAR Embedded Workbench environment of the IAR Systems society. So in order to make it compilable with MSPGCC, some minor changes have to be performed on the SimpliciTI code.

1. `/Components/bsp/mcus/bsp_msp430_defs.h`
   - In part "Unrecognized compiler" (l 102), replace :

     ```
     #error "ERROR: Unknown compiler."
     ```

     by :

     ```
     #ifdef __GNUC__
     #include <io.h>
     #include <signal.h>
     ```

[1] http://focus.ti.com/docs/toolsw/folders/print/simpliciti.html
[2] http://senstools.gforge.inria.fr/doku.php?id=lib:simpliciti
[3] http://senstools.gforge.inria.fr/doku.php?id=lib:simpliciti

```
#include <iomacros.h>
#define __bsp_ISTATE_T__ uint16_t
#define __bsp_ISR_FUNCTION__(f,v) interrupt (v) f(void)
#define __bsp_ENABLE_INTERRUPTS__() eint()
#define __bsp_DISABLE_INTERRUPTS__() dint()
#define __bsp_INTERRUPTS_ARE_ENABLED__() (READ_SR & 0x8)
#define __bsp_GET_ISTATE__() (READ_SR & 0x8)
#define __bsp_RESTORE_ISTATE__(x) st(if((x&GIE))_BIS_SR(GIE);)
#else
#error "ERROR: Unknown compiler."
#endif
```

- In part "Common" (l 137), replace :

```
typedef   signed char     int8_t;
typedef   signed short    int16_t;
typedef   signed long     int32_t;

typedef   unsigned char   uint8_t;
typedef   unsigned short  uint16_t;
typedef   unsigned long   uint32_t;
```

by :

```
#ifndef __GNUC_
typedef   signed char     int8_t;
typedef   signed short    int16_t;
typedef   signed long     int32_t;

typedef   unsigned char   uint8_t;
typedef   unsigned short  uint16_t;
typedef   unsigned long   uint32_t;
#endif
```

2. `/Components/bsp/drivers/code/bsp_generic_buttons.h`

   Replace (l 197) :

   ```
   #error "ERROR: Debounce delay macro is missing."
   ```

   by :

   ```
   #ifndef __GNUC__
   #error "ERROR: Debounce delay macro is missing."
   #endif
   ```

3. `/Components/simpliciti/nwk/nwk_QMgmt.c`

   Replace (l 40) :

   ```
   #include <intrinsics.h>
   ```

   by :

   ```
   #ifndef __GNUC_
   #include <intrinsics.h>
   #endif
   ```

5.2 Using it ported

Just download and extract the `simpliciti-wsn430-v1.1.1.tar.gz` archive, available on SimpliciTI page of the Senstools website[4] (SimpliciTI ported for WSN430). SimpliciTI 1.1.1 ported for WSN430 and compilable with MSPGCC is ready to use.

SimpliciTI source code is in the `/Components` directory, including WSN430 board source code (located in `/Components/bsp/boards`). Examples are stored in the `/Projects/Examples` folder. SimpliciTI official documentations are available in the `/Documents` directory.

[4] http://senstools.gforge.inria.fr/doku.php?id=lib:simpliciti

Chapter 6

Program example

6.1 Introduction

This simple program shows how to implement communication thanks to SimpliciTI. In this program we establish a peer to peer communication between two end devices : a **sender** and a **listener**. These devices are going to exchange frames containing a led number to turn on and a transmitted frame id, in order to demonstrate the communication.

6.2 Description

The sender periodically sends a frame that includes a table of 2 elements called *msg*. The first element is the led number to toggle. In this case it is always the green led. The second one is a number representing the sent frame id. Thus each time the sender sends a frame, this counter is incremented.

The listener is waiting in an infinite loop for incoming frame. As soon as it receives a frame, a callback function is launch that will toggle the green led of the listener device, if the message is valid. As acknowledgement, the listener sends a frame to the sender containing a led number (the red) to turn on and a transmitted frame id.

The sender is also always checking for incoming frame, and when it receives a valid frame from the listener, it calls a callback function checking the message and toggling the red led.

6.3 SimpliciTI APIs usage

The first function to be invoked is the `BSP_Init()`, to initialize the specific target platform.

Then the `SMPL_Init(sRxCallback)` API initializes the radio and the SimpliciTI protocol stack. The `sRxCallback()` is the user callback function.

These two first steps are done in the sender device program code, as well as in the listener device one.

At this step, the sender tries to initiate the communication by executing periodically the SimpliciTI API : `SMPL_Link()`. Once the listener is ready to communicate, it runs the SimpliciTI API : `SMPL_LinkListen()`. In case of success, the two devices are linked.

The last APIs used is this example are `SMPL_Send()` and `SMPL_Receive()`, to send/receive to/from a specified device frames.

6.4 Compiling and executing the example

This example is located `/Projects/Examples/WSN430` folder. It is composed of two main folders, the first one for the sender device files (`/sender`), and the second one for the listener device files (`/listener`). To compile this example you just have to go into them, and executing the `make` command.

Note that SimpliciTI device addresses are very important, to enable them to communicate. Theses addresses are defined in the devices makefile, with the following command and address for instance: `SMPL_NWK_CONFIG  += -DTHIS_DEVICE_ADDRESS="{0x79, 0x56, 0x34, 0x13}"`.

6.5 Example code

6.5.1 Sender code (main_LinkTo.c)

```
#include "bsp.h"
#include "mrfi.h"
#include "nwk_types.h"
#include "nwk_api.h"
#include "bsp_leds.h"
#include "bsp_buttons.h"

//#include "app_remap_led.h"

static void linkTo(void);

void toggleLED(uint8_t);

static uint8_t  sTxTid, sRxTid;
static linkID_t sLinkID1;

/* application Rx frame handler. */
static uint8_t sRxCallback(linkID_t);

#define SPIN_ABOUT_A_SECOND  NWK_DELAY(100)

void main (void)
{
  BSP_Init();


  /* If an on-the-fly device address is generated it must be done before the
   * call to SMPL_Init(). If the address is set here the ROM value will not
   * be used. If SMPL_Init() runs before this IOCTL is used the IOCTL call
   * will not take effect. One shot only. The IOCTL call below is conformal.
   */
#ifdef I_WANT_TO_CHANGE_DEFAULT_ROM_DEVICE_ADDRESS_PSEUDO_CODE
  {
    addr_t lAddr;

    createRandomAddress(&lAddr);
    SMPL_Ioctl(IOCTL_OBJ_ADDR, IOCTL_ACT_SET, &lAddr);
  }
#endif /* I_WANT_TO_CHANGE_DEFAULT_ROM_DEVICE_ADDRESS_PSEUDO_CODE */

  /* This call will fail because the join will fail since there is no Access Point
   * in this scenario. But we don't care -- just use the default link token later.
   * We supply a callback pointer to handle the message returned by the peer.
   */
```

```
  SMPL_Init(sRxCallback);


  /* turn on LEDs alternatively */
  toggleLED(1);
  toggleLED(2);
  toggleLED(3);
  NWK_DELAY(200);
  int i,j;
  for (i = 6; --i >= 0; ) {
    for (j = 1; j<=3 ; j++) {
      toggleLED(j);
      NWK_DELAY(20);
    }
  }


  /* never coming back... */
  linkTo();

  /* but in case we do... */
  while (1) ;
}

static void linkTo()
{
  uint8_t  msg[2], delay = 0;

  while (SMPL_SUCCESS != SMPL_Link(&sLinkID1))
  {
    /* blink red LED, until we link successfully */
    toggleLED(1);
    NWK_DELAY(100);//SPIN_ABOUT_A_SECOND;
  }

  /* we're linked. turn off LEDs. Received messages will toggle the green LED. */
  if (BSP_LED2_IS_ON())
  {
    toggleLED(1);
  }

  if (BSP_LED1_IS_ON())
  {
    toggleLED(2);
  }

  if (BSP_LED1_IS_ON())
  {
    toggleLED(3);
  }

  /* turn on RX. default is RX off. */
  SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_RXON, 0);
```

```
  /* put LED to toggle in the message */
  msg[0] = 2;  /* toggle green */
  while (1)
  {
    SPIN_ABOUT_A_SECOND;
    if (delay > 0x00)
    {
      SPIN_ABOUT_A_SECOND;
    }
    if (delay > 0x01)
    {
      SPIN_ABOUT_A_SECOND;
    }
    if (delay > 0x02)
    {
      SPIN_ABOUT_A_SECOND;
    }

    /* delay longer and longer -- then start over */
    delay = (delay+1) & 0x03;
    /* put the sequence ID in the message */
    msg[1] = ++sTxTid;
    SMPL_Send(sLinkID1, msg, sizeof(msg));
  }
}


void toggleLED(uint8_t which)
{
  switch(which)
   {
    case 1 :
      BSP_TOGGLE_LED1();
      break;
    case 2 :
      BSP_TOGGLE_LED2();
      break;
    case 3 :
      BSP_TOGGLE_LED3();
      break;
   }
  return;
}


/* handle received frames. */
static uint8_t sRxCallback(linkID_t port)
{
  uint8_t msg[2], len, tid;

  /* is the callback for the link ID we want to handle? */
  if (port == sLinkID1)
```

```
  {
    /* yes. go get the frame. we know this call will succeed. */
     if ((SMPL_SUCCESS == SMPL_Receive(sLinkID1, msg, &len)) && len)
     {
       /* Check the application sequence number to detect
        * late or missing frames...
        */
       tid = *(msg+1);
       if (tid)
       {
         if (tid > sRxTid)
         {
           /* we're good. toggle LED in the message */
           toggleLED(*msg);
           sRxTid = tid;
         }
       }
       else
       {
         /* the wrap case... */
         if (sRxTid)
         {
           /* we're good. toggle LED in the message */
           toggleLED(*msg);
           sRxTid = tid;
         }
       }
       /* drop frame. we're done with it. */
       return 1;
     }
  }
  /* keep frame for later handling. */
  return 0;
}
```

6.5.2 Listener code (main_LinkListen.c)

```
#include "bsp.h"
#include "mrfi.h"
#include "nwk_types.h"
#include "nwk_api.h"
#include "bsp_leds.h"
#include "bsp_buttons.h"

//#include "app_remap_led.h"

static void linkFrom(void);

void toggleLED(uint8_t);

static          uint8_t  sRxTid;
static          linkID_t sLinkID2;
static volatile uint8_t  sSemaphore;
```

```
/* Rx callback handler */
static uint8_t sRxCallback(linkID_t);

int main (void)
{
  BSP_Init();

  /* If an on-the-fly device address is generated it must be done before the
   * call to SMPL_Init(). If the address is set here the ROM value will not
   * be used. If SMPL_Init() runs before this IOCTL is used the IOCTL call
   * will not take effect. One shot only. The IOCTL call below is conformal.
   */
#ifdef I_WANT_TO_CHANGE_DEFAULT_ROM_DEVICE_ADDRESS_PSEUDO_CODE
  {
    addr_t lAddr;

    createRandomAddress(&lAddr);
    SMPL_Ioctl(IOCTL_OBJ_ADDR, IOCTL_ACT_SET, &lAddr);
  }
#endif /* I_WANT_TO_CHANGE_DEFAULT_ROM_DEVICE_ADDRESS_PSEUDO_CODE */

  /* This call will fail because the join will fail since there is no Access Point
   * in this scenario. But we don't care -- just use the default link token later.
   * We supply a callback pointer to handle the message returned by the peer.
   */
  SMPL_Init(sRxCallback);

  /* turn on LEDs. */
  toggleLED(2);
  NWK_DELAY(500);
  int i;
  for (i = 10; --i >= 0; ) {
    toggleLED(1);
    toggleLED(2);
    NWK_DELAY(10);
  }
  toggleLED(2);


  /* never coming back... */
  linkFrom();

  /* but in case we do... */
  while (1) ;
}

static void linkFrom()
{
  uint8_t     msg[2], tid = 0;

  /* Turn off one LED so we can tell the device is now listening.
   * Received messages will toggle the other LED.
```

```
   */
  //toggleLED(1);

  /* listen for link forever... */
  while (1)
    {
      if (SMPL_SUCCESS == SMPL_LinkListen(&sLinkID2))
{
  break;
}
      /* Implement fail-to-link policy here. otherwise, listen again. */
    }
  /* we're linked. turn off red LED. Received messages will toggle the green LED. */
  if (BSP_LED2_IS_ON())
    {
      toggleLED(2);
    }

  if (BSP_LED1_IS_ON())
    {
      toggleLED(1);
    }

  /* turn on LED1 on the peer in response to receiving a frame. */
  *msg = 0x01;  /* toggle red led */

  /* turn on RX. default is RX off. */
  SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_RXON, 0);

   while (1)
   {
     /* Wait for a frame to be received. The Rx handler, which is running in
      * ISR thread, will post to this semaphore allowing the application to
      * send the reply message in the user thread.
      */
     if (sSemaphore)
     {
       *(msg+1) = ++tid;
       SMPL_Send(sLinkID2, msg, 2);

       /* Reset semaphore. This is not properly protected and there is a race
        * here. In theory we could miss a message. Good enough for a demo, though.
        */
       sSemaphore = 0;
     }
   }
}


void toggleLED(uint8_t which)
{
  switch(which)
   {
```

```
    case 1 :
      BSP_TOGGLE_LED1();
      break;
    case 2 :
      BSP_TOGGLE_LED2();
      break;
    case 3 :
      BSP_TOGGLE_LED3();
      break;
   }
  return;
}


/* handle received messages */
static uint8_t sRxCallback(linkID_t port)
{
  uint8_t msg[2], len, tid;

  /* is the callback for the link ID we want to handle? */
  if (port == sLinkID2)
  {
    /* yes. go get the frame. we know this call will succeed. */
     if ((SMPL_SUCCESS == SMPL_Receive(sLinkID2, msg, &len)) && len)
     {
       /* Check the application sequence number to detect
        * late or missing frames...
        */
       tid = *(msg+1);
       if (tid)
       {
         if (tid > sRxTid)
         {
           /* we're good. toggle LED */
           toggleLED(*msg);
           sRxTid = tid;
         }
       }
       else
       {
         /* wrap case... */
         if (sRxTid)
         {
           /* we're good. toggle LED */
           toggleLED(*msg);
           sRxTid = tid;
         }
       }
       /* Post to the semaphore to let application know so it sends
        * the reply
        */
       sSemaphore = 1;
       /* drop frame. we're done with it. */
```

```
      return 1;
    }
  }
  /* keep frame for later handling */
  return 0;
}
```